Question: all_divisors

```cpp
#include <iostream>
#include <cmath>
#include <bits/stdc++.h>
using namespace std;

void get_all_divisors()
{
    // this has t.c. O(N)

    int num;
    int count = 0;
    cout << "Enter num :";
    cin >> num;

    if (num <= 0)
    {
        cout << "Please provide positive integer only" << endl;
        return;
    }

    cout << "The divisors of " << num << " are : " << endl;

    for (int i = 1; i <= num; i++)
    {

        if (num % i == 0)
        {
            cout << i << " ";
            count += 1;
            cout << endl;
        }
    }
    cout << "The number of divisors of " << num << " are : " << count << endl;
}

void get_div()
{
    // reducing the t.c.
    int n;
    cin >> n;
    vector<int> ls;

    // for (int i = 1; i <= sqrt(n); i++) here since sqrt itself is a function and every time it gets called it will
    // here, t.c. is O(sqrt(n))
    for (int i = 1; i * i <= n; i++) // another way for using sqrt
    {
        if (n % i == 0)
        {
            ls.push_back(i);
            if ((n / i) != i)
            {
                ls.push_back(n / i);
            }
        }
    }

    // here, t.c. is O(no of factors + log(no of factors))
    sort(ls.begin(), ls.end());

    // O(number of factors)
    for (auto it : ls)
        cout << it << " ";
}

int main()
{
    int t;
    cout << "Enter number of test cases : ";
    cin >> t;
    for (int i = 0; i < t; i++)
    {
        // get_all_divisors();
        get_div();
    }
    return 0;
}
```

Question: armstrong_num

```
#include <iostream>
#include <cmath>
using namespace std;

void find_arm3()
{
    // This for numbers with 3 digits only
    int num;
    int lastDigit, sum = 0;
    // int sum = 0;
    int reverseNumber = 0;
    cout << "enter num : ";
    cin >> num;
    while (num != 0)
    {
        lastDigit = num % 10;
        sum += lastDigit * lastDigit * lastDigit;
        num = num / 10;
        reverseNumber = (reverseNumber * 10) + lastDigit;
    }
    cout << "Rev" << reverseNumber << endl;
    cout << "sum" << sum << endl;
    // cout << sum;
    // if (num == sum)
    // {
    //     cout << sum;
    // }
    // cout << endl;
}

void find_armN()
{
    int num;
    int originalNum, lastDigit, sum = 0, numDigits = 0;
    cout << "Enter num: ";
    cin >> num;

    // Store the original number to compare later
    originalNum = num;

    // Step 1: Calculate the number of digits in the number
    while (num != 0)
    {
        numDigits++;
        num = num / 10;
    }

    // Step 2: Reset num to original value and calculate the sum of digits raised to the power of numDigits
    num = originalNum;
    sum = 0; // Reset sum to 0

    while (num != 0)
    {
        lastDigit = num % 10;              // Get the last digit
        sum += pow(lastDigit, numDigits); // Add the digit raised to the power of numDigits
        num = num / 10;                    // Remove the last digit
    }

    // Output results
    cout << "Sum of powers of digits: " << sum << endl;

    // Check if the number is an Armstrong number
    if (sum == originalNum)
    {
        cout << originalNum << " is an Armstrong number." << endl;
    }
    else
    {
        cout << originalNum << " is NOT an Armstrong number." << endl;
    }
}
```

```
int main()
{
    int t;
    cout << "Enter number of test cases : ";
    cin >> t;
    for (int i = 0; i < t; i++)
    {
        find_armN();
    }
}
```

Question: count_digit

```
#include <iostream>
using namespace std;
void countDig()
{
    int N;
    cout << "Enter the number N: ";
    cin >> N;
    int count = 0;
    while (N > 0)
    {
        int lastDigit = N % 10; // if N = 5546, this will give 6
        cout << "Last digit : " << lastDigit << endl;
        N = N / 10; // if N = 5546, this will give 554
        count += 1;
        cout << "The number of digits in number is : " << count;
    }
    cout << endl;
}

int main()
{
    int t;
    cout << "Enter number of test cases : ";
    cin >> t;
    for (int i = 0; i < t; i++)
    {
        countDig();
    }
}
```

Question: find_occurence_in_array

```
#include <iostream>
using namespace std;

int findNumOfOccurr(int number, int arr[], int n)
{
    // here for finding count for every number, N steps is performed so t.c. is O(N)
    // when number of numbers becomes very large, for eg, Q, then t.c. would be O(Q * N)

    int count = 0;
    for (int i = 0; i < n; i++)
    {
        if (arr[i] == number)
            count += 1;
    }
    return count;
}

int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
        cin >> arr[i];
    cout << findNumOfOccurr(4, arr, n);
}
```

```
--------------------------------------------------------------------------------

Question: get_prime

#include <iostream>
#include <cmath>
#include <bits/stdc++.h>
using namespace std;

bool check_prime()
{
    int n;
    cin >> n;

    if (n <= 1)
        return false;

    for (int i = 2; i * i <= n; i++)
    {
        if (n % i == 0)
            return false;
    }

    return true; // n is prime if no divisors were found
}

int main()
{
    int t;
    cout << "Enter number of test cases: ";
    cin >> t;

    for (int i = 0; i < t; i++)
    {
        if (check_prime())
            cout << "prime" << endl;
        else
            cout << "not prime" << endl;
    }

    return 0;
}

--------------------------------------------------------------------------------

Question: hcf

#include <iostream>
#include <cmath>
#include <bits/stdc++.h>
using namespace std;

void get_hcf()
{
    int n1, n2, gcd = 1;
    cout << "Enter n1" << endl;
    cin >> n1;
    cout << "Enter n2" << endl;
    cin >> n2;

    // t.c. : O(min(n1,n2))
    for (int i = 1; i <= min(n1, n2); i++)
    {
        if (n1 % i == 0 && n2 % i == 0)
        {
            gcd = i;
            cout << gcd << endl; // this prints all common divisors
        }
    }
    cout << gcd << endl; // this gives the actual gcd/hcf.
}

// to reduce t.c. further, use euclidean algorithm which states that
// gcd(a,b) = gcd(a-b, b) where a > b
// but there can be catch in many case

// so use this maybe for better t.c.
// gcd(a,b) == gcd(a%b, b) where a > b go on doing this step, untill one of them becomes 0.
// when one number becomes 0, the other will be gcd

// So now, t.c. becomes O(log■(min(a,b))) where x = ϕ

int main()
{
    int t;
    cout << "Enter number of test cases : ";
    cin >> t;
    for (int i = 0; i < t; i++)
    {
        // get_all_divisors();
        get_hcf();
    }
    return 0;
}
```

Question: reverse_number

```
#include <iostream>
using namespace std;
void reverseNum()
{
    int N;
    int reverseNumber = 0;
    cout << "Enter the number N: ";
    cin >> N;
    int count = 0;
    while (N > 0)
    {
        int lastDigit = N % 10; // if N = 5546, this will give 6
        cout << "Last digit : " << lastDigit << endl;
        N = N / 10; // if N = 5546, this will give 554

        // this step allows the next number to get added into the previous number
        // for eg, the first result is 9 and second is 8
        // so to get 8 after 9, multiply 9 by 10 and add 8 to it
        reverseNumber = (reverseNumber * 10) + lastDigit;
        cout << "The reverse number is : " << reverseNumber;
    }
    cout << endl;
}

int main()
{
    int t;
    cout << "Enter number of test cases : ";
    cin >> t;
    for (int i = 0; i < t; i++)
    {
        reverseNum();
    }
}
```

Question: 3sum

// here given an array we need to find all the triplets that sum to zero

```
#include <bits/stdc++.h>
using namespace std;

// brute force approach
vector<vector<int>> triplet(int n, vector<int> &num)
{
    // tc : O(N^3) + log(no. of unique triplets) and sc : 2 * O(no. of triplets)

    set<vector<int>> st;
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            for (int k = j + 1; k < n; k++)
            {
                // here we are checking if the sum of num[i] + num[j] + num[k] == 0
                if (num[i] + num[j] + num[k] == 0)
                {
                    vector<int> temp = {num[i], num[j], num[k]};
                    sort(temp.begin(), temp.end());
                    st.insert(temp);
                }
            }
        }
    }
    vector<vector<int>> ans(st.begin(), st.end());
    return ans;
}

// better approach
// in last approach we are using 3 loops, we can reduce it to 2 loops
// there it was num[i] + num[j] + num[k] == 0, so here, num[k] = -(num[i] + num[j])
vector<vector<int>> triplet_2(int n, vector<int> &num)
{
    // tc : O(N^2) * log(size of set) and sc : 2 * O(no. of triplets) + O(N)
    set<vector<int>> st;
    for (int i = 0; i < n; i++)
    {
        set<int> hashset;
        for (int j = i + 1; j < n; j++)
        {
            int third = -(num[i] + num[j]);
            if (hashset.find(third) != hashset.end())
            {
                vector<int> temp = {num[i], num[j], third};
                sort(temp.begin(), temp.end());
                st.insert(temp);
            }
            hashset.insert(num[j]);
        }
    }
    vector<vector<int>> ans(st.begin(), st.end());
    return ans;
}

// optimal approach
// here we will sort the array and then use 2 pointers approach
vector<vector<int>> triplet(int n, vector<int> &num)
{
    vector<vector<int>> ans;
    sort(num.begin(), num.end());
    for (int i = 0; i < n; i++)
    {
        if (i > 0 && num[i] == num[i - 1])
            continue; // skip duplicates for the first element
        int j = i + 1, k = n - 1;
        while (j < k)
        {
            int sum = num[i] + num[j] + num[k];
            if (sum < 0)
            {
                j++;
            }
            else if (sum > 0)
            {
                k--;
            }
            else
            {
                vector<int> temp = {num[i], num[j], num[k]};
                ans.push_back(temp);
                j++;
                k--;
                while (j < k && num[j] == num[j - 1])
                    j++;
                while (j < k && num[k] == num[k + 1])
                    k--;
            }
        }
    }
    return ans;
```

```
}
```

Question: 4sum

```
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> quadruplets(int n, vector<int> &num)
{
    set<vector<int>> st;
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            for (int k = j + 1; k < n; k++)
            {
                for (int l = k + 1; l < n; l++)
                {
                    // here we are checking if the sum of num[i] + num[j] + num[k] + num[l] == 0
                    if (num[i] + num[j] + num[k] + num[l] == 0)
                    {
                        vector<int> temp = {num[i], num[j], num[k], num[l]};
                        sort(temp.begin(), temp.end());
                        st.insert(temp);
                    }
                }
            }
        }
    }
    vector<vector<int>> ans(st.begin(), st.end());
    return ans;
}

// better approach
// in last approach we are using 4 loops, we can reduce it to 3 loops
vector<vector<int>> fourSum(vector<int> &nums, int target)
{
    // total tc : O(N^3 * log(size of set)) and sc : 2 * O(no. of quadruplets) + O(N)
    int n = nums.size();
    set<vector<int>> st;
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            set<long long> hashset;
            for (int k = j + 1; j < n; j++)
            {
                long long fourth = target - nums[i] - nums[j] - nums[k];
                // this is to check if fourth is present in hashset
                if (hashset.find(fourth) != hashset.end())
                {
                    vector<int> temp = {nums[i], nums[j], nums[k], (int)fourth};
                    sort(temp.begin(), temp.end());
                    st.insert(temp);
                }
                // nums[k] is inserted into hashset so that, in subsequent iterations, you can quickly check if the
                // to complete the quadruplet has already been seen. This is a common technique for reducing nested
                hashset.insert(nums[k]);
            }
        }
    }
    vector<vector<int>> ans(st.begin(), st.end());
    return ans;
}

// optimal approachthis is similar to 3 sum problem
vector<vector<int>> fourSumOptimal(vector<int> &nums, int target)
{
    int n = nums.size();
    vector<vector<int>> ans;
    sort(nums.begin(), nums.end()); // sorting the array to use two pointers approach
    for (int i = 0; i < n; i++)
    {
        if (i > 0 && nums[i] == nums[i - 1])
```

```
                continue; // to avoid duplicates
            for (int j = i + 1; j < n; j++)
            {
                if (j != (i + 1) && nums[j] == nums[j - 1])
                    continue; // to avoid duplicates
                int k = j + 1, l = n - 1;
                while (k < l)
                {
                    long long sum = nums[i];
                    // elements at indices j, k, and l are added to the sum in this way, so that integer overflow does n
                    sum += nums[j];
                    sum += nums[k];
                    sum += nums[l];
                    if (sum == target)
                    {
                        vector<int> temp = {nums[i], nums[j], nums[k], nums[l]};
                        ans.push_back(temp);
                        k++;
                        l--;
                        // this is to skip duplicates in the current iteration
                        // to avoid duplicates
                        while (k < l && nums[k] == nums[k - 1])
                            k++;
                        while (k < l && nums[l] == nums[l + 1])
                            l--;
                        k++;
                        l--;
                    }
                    else if (sum < target)
                        k++;
                    else
                        l--;
                }
            }
        }
        return ans;
    }

    int main()
    {
        int n;
        cout << "Enter number of elements: ";
        cin >> n;
        /*
            Enter number of elements: 6
            Enter 6 integers: 1 0 -1 0 2 -2

            ans : Quadruplets with sum 0 are:
            -2 -1 1 2
            -2 0 0 2
            -1 0 0 1

        */

        vector<int> num(n);
        cout << "Enter " << n << " integers: ";
        for (int i = 0; i < n; ++i)
        {
            cin >> num[i];
        }

        vector<vector<int>> result = quadruplets(n, num);

        cout << "Quadruplets with sum 0 are:\n";
        for (const auto &quad : result)
        {
            for (int val : quad)
            {
                cout << val << " ";
            }
            cout << "\n";
        }

        return 0;
    }
```

Question: basic

```
/*

contains elements of same data type.
if declared inside int main() then garbage value will be stored inside array if nothing else is initialized.
and if declared globally then zeros will be stored inside array if nothing is initialized.


max size of array declared inside int main() is 10^6 i.e. arr[10^6]
and if declared globally then max size is 10^7, i.e. arr[10^7]

Accessing an array :  By index. starts from 0 ends at n-1
Accessing arrays by address is not possible so we access it by index.
*/
```

Question: check_sorted

```cpp
#include <bits/stdc++.h>
using namespace std;

int isSorted(int n, vector<int> a)
{
    for (int i = 1; i < n; i++)
    {
        if (a[i] < a[i - 1])
        {
            return false;
        }
    }
    return true;
}

int main()
{
    int n;
    cin >> n;
    vector<int> a(n);
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];
    }

    if (isSorted(n, a))
    {
        cout << "The array is sorted." << endl;
    }
    else
    {
        cout << "The array is not sorted." << endl;
    }

    return 0;
}
```

Question: count_inversions_in_an_array

```cpp
// in this sum, we have to find pairs in array such that the number on the left side of array is greater than number
// eg : vector<int> arr = {5, 3, 2, 4, 1}; the count of such pairs is 8. like (5,3) is a pair but (3,4) isn't coz 3

#include <bits/stdc++.h>
using namespace std;

// brute force approach, tc : O(N^2)
int find_pairs(vector<int> arr)
{
    int n = arr.size();
    int count_brute = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (arr[i] > arr[j])
                count_brute++;
        }
    }
    return count_brute;
}

int merge(vector<int> &arr, int low, int mid, int high)
{
    int count = 0;
    vector<int> temp; // Temporary array to store merged result
    // Space Complexity: O(n), where n = high - low + 1

    // Left subarray -> [low....mid]
    // Right subarray -> [mid+1....high]

    int left = low;
    int right = mid + 1;

    // Merge the two sorted halves
    while (left <= mid && right <= high)
    {
        // Time Complexity (in total for the entire merge step across the array): O(n)
        if (arr[left] <= arr[right]) // left is smaller
        {
            temp.push_back(arr[left]);
            left++;
        }
        // right is smaller
        else
        {
            temp.push_back(arr[right]);
            count += (mid - left + 1);
            right++;
        }
    }

    // Copy any remaining elements from the left subarray
    while (left <= mid)
    {
        temp.push_back(arr[left]);
        left++;
        // Worst case: O(n) if all elements are in left subarray
    }

    // Copy any remaining elements from the right subarray
    while (right <= high)
    {
        temp.push_back(arr[right]);
        right++;
        // Worst case: O(n) if all elements are in right subarray
    }

    // Copy merged elements back into original array
    for (int i = low; i <= high; i++)
    {
        arr[i] = temp[i - low];
        // Time Complexity: O(n)
    }

    // Total Time Complexity of merge() = O(n), where n = high - low + 1
    // Total Space Complexity of merge() = O(n)
    return count;
}

void mS(vector<int> &arr, int low, int high)
{
    // Base case: when only one element is left
    if (low == high)
        return;

    // Calculate the middle point
    int mid = (low + high) / 2;

    // Recursively sort the left half
    mS(arr, low, mid);

    // Recursively sort the right half
    mS(arr, mid + 1, high);

    // Merge the sorted halves
    merge(arr, low, mid, high);

    // Time Complexity per level: O(n) (from merge)
    // Number of levels (recursive depth): O(log n)
    // So, total Time Complexity: O(n log n)

    // Space Complexity: O(log n) due to recursive call stack (implicit)
    // Plus O(n) for temp arrays in each merge
    // So, overall space:
    // - Auxiliary/Recursive stack: O(log n)
    // - Temporary arrays: O(n) (not per call, total across all levels)
}

void mergeSort(vector<int> &arr, int low, int high)
{
    if (low >= high)
        return;
    int mid = (low + high) / 2;
    mergeSort(arr, low, mid);
    mergeSort(arr, mid + 1, high);
    merge(arr, low, mid, high);
}

int numberOfInversions(vector<int> &a, int n)
{
    mergeSort(a, 0, n - 1);
    // TO FIX : TODO
    // return count;
}

int main()
{
    vector<int> arr = {5, 3, 2, 4, 1};
    // int result = find_pairs_better(arr);
    // cout << result << endl;
}
```

Question: count_subarray_sums_equals_k

```
// #include <bits/stdc++.h>
// using namespace std;

// int find_num_subarray_Sum_k(int arr[], int n, int target)
// {
//     int count = 0;
//     for (int i = 0; i < n; i++)
//     {
//         for (int j = i; j < n; j++)
//         {
//             int sum = 0;
//             for (int k = i; k <= j; k++)
//             {
//                 sum = sum + arr[k];
//             }
//             if (sum == target)
//                 count++;
//         }
//     }
//     return count;
// }

// int main()
// {
//     int arr[] = {1, 5, -2, 4, 1, 2, 1, 1, 1};
//     int n = sizeof(arr) / sizeof(arr[0]);
//     int target = 3;
//     cout << find_num_subarray_Sum_k(arr, n, target) << endl; // Output: 2
//     return 0;
// }
```

```cpp
#include <bits/stdc++.h>
using namespace std;

// returns the number of times sum is reached in the subarray
int find_num_subarray_Sum_k(int arr[], int n, int target)
{
    int count = 0;
    for (int i = 0; i < n; i++) // Start of the subarray
    {
        for (int j = i; j < n; j++) // End of the subarray
        {
            int sum = 0;
            for (int k = i; k <= j; k++) // Sum the subarray
            {
                sum = sum + arr[k];
            }
            cout << "Subarray: ";
            for (int k = i; k <= j; k++) // Print the subarray
            {
                cout << arr[k] << " ";
            }
            cout << "Sum: " << sum << endl;

            if (sum == target)
                count++;
        }
    }
    return count;
}
int find_num_subarray_Sum_k_better(int arr[], int n, int target)
{
    int count = 0;
    for (int i = 0; i < n; i++) // Start of the subarray
    {
        int sum = 0;
        for (int j = i; j < n; j++) // End of the subarray
        {
            sum = sum + arr[j];

            cout << "Subarray: ";
            for (int j = i; j <= i; j++) // Print the subarray
            {
                cout << arr[j] << " ";
            }
            cout << "Sum: " << sum << endl;

            if (sum == target)
                count++;
        }
    }
    return count;
}

int findAllSubarraysWithGivenSumOptimal(vector<int> &arr, int k)
{
    // Step 1: Declare a map to store the frequency of prefix sums
    map<int, int> mpp;

    // Step 2: Initialize the map with {0: 1}.
    // This is because a prefix sum of 0 means a valid subarray can start from index 0 itself if the sum equals `k`.
    mpp[0] = 1;

    // Step 3: Initialize variables for prefix sum and the result count
    int preSum = 0, count = 0;

    // Step 4: Iterate through the array to calculate prefix sums and count subarrays with sum `k`
    for (int i = 0; i < arr.size(); i++)
    {

        // Step 5: Add the current element to the prefix sum
        preSum += arr[i];

        // Step 6: Calculate the difference (remove) between current prefix sum and target sum `k`
        // This represents the sum that should have existed earlier in the array for the subarray sum to be `k`
        int remove = preSum - k;
        // Step 7: If this difference (remove) exists in the map, it means we've seen a prefix sum before
        // that, when subtracted from the current prefix sum, gives the target sum `k`
        count += mpp[remove];

        // Step 8: Update the map with the current prefix sum.
        // It helps to count how many times we've seen this prefix sum
        mpp[preSum] += 1;
    }

    // Step 9: Return the final count of subarrays with sum `k`
    return count;
}

int main()
{
    int arr[] = {1, 5, -2, 4, 1, 2, 1, 1, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 3;
    cout << "Number of subarrays: " << find_num_subarray_Sum_k_better(arr, n, target) << endl;
    return 0;
}
```

Question: DP_buy_sell_stocks_on_bestTime

```
#include <bits/stdc++.h>
using namespace std;

int maximumProfit(vector<int> &prices)
{
    int mini = prices[0];
    int maxProfit = 0;
    int n = prices.size();
    for (int i = 0; i < n; i++)
    {
        int cost = prices[i] - mini;
        maxProfit = max(maxProfit, cost);
        mini = min(mini, prices[i]);
    }
    return maxProfit;
}

int maximumProfitAllSteps(vector<int> &prices)
{
    int mini = prices[0]; // Initialize the minimum price to the first price
    int maxProfit = 0;    // Start with 0 profit
    int n = prices.size();

    cout << "Starting maximum profit calculation..." << endl;
    cout << "Initial minimum price: " << mini << endl;

    // Loop through each price
    for (int i = 0; i < n; i++)
    {
        // Calculate the potential profit if sold at the current price
        int cost = prices[i] - mini;

        // Update the maximum profit if the current profit is larger
        maxProfit = max(maxProfit, cost);

        // Update the minimum price encountered so far
        mini = min(mini, prices[i]);

        // Print the important intermediate values
        cout << "At day " << i + 1 << ", price: " << prices[i] << endl;
        cout << "Current minimum price: " << mini << endl;
        cout << "Current profit: " << cost << endl;
        cout << "Max profit so far: " << maxProfit << endl;
    }

    return maxProfit; // Return the maximum profit

    /*
    n = 6
    arr = 7 1 5 3 6 4
    Starting maximum profit calculation...
    Initial minimum price: 7
    At day 1, price: 7
    Current minimum price: 7
    Current profit: 0
    Max profit so far: 0
    At day 2, price: 1
    Current minimum price: 1
    Current profit: -6
    Max profit so far: 0
    At day 3, price: 5
    Current minimum price: 1
    Current profit: 4
    Max profit so far: 4
    At day 4, price: 3
    Current minimum price: 1
    Current profit: 2
    Max profit so far: 4
    At day 5, price: 6
    Current minimum price: 1
    Current profit: 5
    Max profit so far: 5
    At day 6, price: 4
    Current minimum price: 1
    Current profit: 3
    Max profit so far: 5
    5

        */
}

int main()
{
    int n;
    cin >> n;
    vector<int> prices(n);
    for (int i = 0; i < n; i++)
    {
        cin >> prices[i];
    }
    cout << maximumProfitAllSteps(prices) << endl;
    return 0;
}
```

Question: find_missing_and_repeated_number

```
// in this problem, given an array of integers from 1 to n, find missing and repeated numbers in array

#include <bits/stdc++.h>
using namespace std;

// brute force approach
vector<int> find_missing_and_repeated(vector<int> &arr)
{
    int n = arr.size();
    int repeated = -1, missing = -1;
    for (int i = 1; i <= n; i++)
    {
        int count = 0;
        for (int j = 0; j < n; j++)
        {
            if (arr[j] == i)
                count++;
        }
        if (count == 2)
            repeated = i;
        else if (count == 0)
            missing = i;
        if (repeated != -1 && missing != -1)
        {
            break;
        }
    }
    return {repeated, missing};
}
```

```
// better approach using hash array, here tc : O(n) + O(n) = O(2n) and sc : O(n) for storing elements in hash array
vector<int> find_missing_and_repeated_better(vector<int> arr)
{
    int n = arr.size();
    int hash[n + 1] = {0}; // created a hash array of size n+1(where n is size of given array) and initialized all v
    for (int i = 0; i < n; i++)
    {
        hash[arr[i]]++; // increasing the count of numbers in array (eg : if 7 is found twice in array, then count s
    }

    int repeating = -1, missing = -1;
    // TODO : why above condition till n-1 and here n
    for (int i = 1; i <= n; i++)
    {
        if (hash[i] == 2)
            repeating = i;
        else if (hash[i] == 0)
            missing = i;

        if (repeating != -1 && missing != -1)
        {
            break;
        }
    }
    return {repeating, missing};
}

// optimal solution includes 2 approaches, one using basic maths and other using XOR(which is termed as hard here)
// in maths approach, we will create two equations. first by adding all the numbers of given array and subtracting i
// here we will consider x and y, x is the repeated number and y is the missing one. so first eqn is created.
// then for second eqn., we all add squares of all numbers of given array and subtract it with squares of numbers ti
// solving both, will give value of x and y which is what we want.

vector<int> find_missing_and_repeated_optimal_maths(vector<int> arr)
{
    // here tc : O(n) and sc : O(1)
    long long n = arr.size();
    // S - Sn = x - y
    // S2 - S2n = x^2 - y^2 = (x+y)(x-y)

    long long Sn = (n * (n + 1)) / 2;
    long long S2n = (n * (n + 1) * (2 * n + 1)) / 6;
    long long S = 0, S2 = 0;

    for (int i = 0; i < n; i++)
    {
        S += arr[i];                                // sum of all elements of given array
        S2 += (long long)arr[i] * (long long)arr[i]; // sum of squares of all numbers given in array
    }

    long long val1 = S - Sn; // x - y
    long long val2 = S2 - S2n;
    val2 = val2 / val1; // x + y
    long long x = (val1 + val2) / 2;
    long long y = x - val1;
    return {(int)x, (int)y};
}

vector<int> find_missing_and_repeated_optimal_xor(vector<int> arr)
{
    long long n = arr.size();
    int xr = 0;
    for (int i = 0; i < n; i++)
    {
        xr = xr ^ arr[i];
        xr = xr ^ (i + 1);
    }

    int bitNo = 0;
    while (1)
    {
        if ((xr & (1 << bitNo)) != 0)
        {
            break;
        }
        bitNo++;
    }

    // int number = xr & ~(xr - 1); this is an alternative for above while loop
    int zero = 0;
    int one = 0;
    for (int i = 0; i < n; i++)
    {
        // part of 1 club, i.e., number ka sabse pehla digit 1 hai in terms of bits. eg : 4 is written as 1 0 0
        if ((arr[i] & (1 << bitNo)) != 0)
        {
            one = one ^ arr[i];
        }

        // part of 0 club, i.e., number ka sabse pehla digit 0 hai in terms of bits. eg : 3 is written as 0 1 1
        else
        {
            zero = zero ^ arr[i];
        }
    }

    for (int i = 1; i <= n; i++)
    {
        // part of 1 club
        if ((i & (1 << bitNo)) != 0)
        {
            one = one ^ i;
        }

        // part of zero club
        else
        {
            zero = zero ^ i;
        }
    }

    int count = 0;
    for (int i = 0; i < n; i++)
    {
        if (arr[i] == zero)
            count++;
    }

    if (count == 2)
        return {zero, one};
    return {one, zero};
}

int main()
{
    vector<int> arr = {1, 2, 2, 4, 5};
    vector<int> result = find_missing_and_repeated(arr);
    cout << "Repeated: " << result[0] << ", Missing: " << result[1] << endl;
    return 0;
}
```

Question: find_missing_in_array

```
#include <bits/stdc++.h>
using namespace std;

int find_missing(int arr[], int n)
{
    // tc : O(N*N) = O(N^2) & sc : O(1)
    for (int i = 1; i < n; i++)
    {
        int flag = 0;
        for (int j = 0; j < n - 1; j++)
        {
            if (arr[j] == i)
            {
                flag = 1;
                break;
            }
        }
        if (flag == 0)
        {
            return i;
        }
    }
}

int find_missing_better(int arr[], int n)
{
    int hash[n + 1] = {0};      // sc : O(N) coz array on N size is used
    for (int i = 0; i < n; i++) // tc: O(N)
    {
        hash[arr[i]] = 1;
    }

    for (int i = 1; i < n; i++) // tc : O(N)
    {
        if (hash[i] == 0)
            return i;
    }
}

int find_missing_optimal1(int arr[], int n)
{
    int sum = n * (n + 1) / 2; // sc : O(1)
    int s2 = 0;
    for (int i = 0; i < n - 1; i++) // tc : O(N)
    {
        s2 += arr[i];
    }
    return sum - s2;
}

int find_missing_zor_optimal(int arr[], int n)
{
    /*
    int XOR1 = 0;
    for (int i = 1; i < n; i++)
    {
        XOR1 = XOR1 ^ i;
        int XOR2 = 0;


        for (int i = 0; i < n - 1; i++)
        {
            XOR2 = XOR2 ^ arr[i];
        }
        return XOR1 ^ XOR2;
    }
    */

    int xor1 = 0;
    int xor2 = 0;
    for (int i = 0; i < n - 1; i++)
    {
        xor2 = xor2 ^ arr[i];
        xor1 = xor1 ^ (i + 1);
    }
    xor1 = xor1 ^ n;
    return xor1 ^ xor2;
}

int main()
{
    int n;
    cin >> n;
    int arr[n - 1];                 // sc : O(1)
    for (int i = 0; i < n - 1; i++) // tc : O(N)
    {
        cin >> arr[i];
    }
    cout << find_missing_zor_optimal(arr, n);
    return 0;
}
```

Question: intersection_sorted_arrays

```
#include <bits/stdc++.h>
using namespace std;

vector<int> findArrayIntersection(vector<int> &A, int n, vector<int> &B, int m)
{
    // tc : O(N1 * N2) & sc : O(N2)
    vector<int> ans;
    int vis[m] = {0};
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (A[i] == B[j] && vis[j] == 0)
            {
                ans.push_back(A[i]);
                vis[j] = 1;
                break;
            }
            if (B[j] > A[i])
                break;
        }
    }
    return ans;
}

vector<int> findArrayIntersectionOptimal(vector<int> &A, int n, vector<int> &B, int m)
{
    // tc : O(N1+N2)
    // sc : O(1)

    int i = 0, j = 0;
    vector<int> ans;
    while (i < n && j < m)
    {
        if (A[i] < B[j])
        {
            i++;
        }
        else if (B[j] < A[i])
        {
            j++;
        }
        else
        {
            ans.push_back(A[i]);
            i++;
            j++;
        }
    }
    return ans;
}

int main()
{
    int n, m;
    cin >> n >> m;

    vector<int> A(n), B(m);

    for (int i = 0; i < n; i++)
    {
        cin >> A[i];
    }

    for (int i = 0; i < m; i++)
    {
        cin >> B[i];
    }

    vector<int> result = findArrayIntersectionOptimal(A, n, B, m);

    for (int i : result)
    {
        cout << i << " ";
    }
    cout << endl;

    return 0;
}
```

Question: largest_element

```
#include <bits/stdc++.h>
using namespace std;

int find_largest()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    int largest = arr[0];
    for (int i = 0; i < n; i++)
    {
        if (arr[i] > largest)
            largest = arr[i];
    }
    return largest;
}

int main()
{
    cout << find_largest() << endl;
}
```

Question: leaders_in_array

```
#include <bits/stdc++.h>
using namespace std;

// leader means a number which is greater than all numbers on its right side in an array
list<int> find_leader(int arr[], int n)
{
    list<int> ans;
    for (int i = 0; i < n; i++)
    {
        bool leader = true;
        for (int j = i + 1; j < n; j++)
        {
            if (arr[j] > arr[i])
            {
                leader = false;
                break;
            }
        }
        if (leader == true)
            ans.push_back(arr[i]);
    }
    return ans;
}

vector<int> find_leaders_optimal(vector<int> &a)
{
    // O(N)
    vector<int> ans;
    int maxi = INT_MIN;
    int n = a.size();

    // O(N)
    for (int i = n - 1; i >= 0; i--)
    {
        if (a[i] > maxi)
        {
            ans.push_back(a[i]);
        }
        // keeps track of right maximum
        maxi = max(maxi, a[i]);
    }

    // O(N log N)
    sort(ans.begin(), ans.end());
    return ans;
}

int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    list<int> leaders = find_leader(arr, n);
    for (int leader : leaders)
    {
        cout << leader << " ";
    }
    cout << endl;

    return 0;
}
```

Question: left_rot_by_d_places

```
#include <bits/stdc++.h>
using namespace std;

void left_rot_by_d(int arr[], int n, int d)
{
    d = d % n;
    int temp[d];

    // O(d)
    for (int i = 0; i < d; i++)
    {
        temp[i] = arr[i];
    }

    // O(n-d)
    for (int i = d; i < n; i++)
    {
        arr[i - d] = arr[i];
    }

    // O(d)
    for (int i = n - d; i < n; i++)
    {
        arr[i] = temp[i - (n - d)];
    }

    // Total t.c. is : O(d) + O(n-d) + O(d) = O(n+d)
    // extra space used is O(d) which is due to temp array
    // overall s.c is O(n) + O(d)
}

void left_rot_by_d_optimal(int arr[], int n, int d)
{
    // here we remove the extra space used due to temp array
    // we perform following reverses, to avoid making use of temp array

    // reverse (a, a+d)  |   O(d)
    // reverse(a+d, a+n) |   O(n-d)
    // reverse(a, a+n)   |   O(n)
    // so total t.c. here is : O(2n)
    // but extra space used is O(1) and overall s.c. is O(n) for n sized array

    /*
    Manual reverse function :
    void reverse(int arr[], int start, int end){
        while(start <= end){
            int temp = arr[start];
            arr[start] = arr[end];
            arr[end] = temp;
            start++;
            end--;
    }
    }
    */

    // Original array : [1, 8, 6, 3, 2, 5, 9]
    reverse(arr, arr + d);     // [6, 8, 1, 3, 2, 5, 9]
    reverse(arr + d, arr + n); // [6, 8, 1, 9, 5, 2, 3]
    reverse(arr, arr + n);     // [3, 2, 5, 9, 1, 8, 6]
    d = d % n;
}

int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    int d;
    cin >> d;
    left_rot_by_d_optimal(arr, n, d);
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Question: linear_search

```
#include <bits/stdc++.h>
using namespace std;

int linear_search_num(int arr[], int n, int num)
{
    for (int i = 0; i < n; i++)
    {
        if (arr[i] == num)
        {
            return i;
        }
    }
    return -1;
}

int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    int num;
    cin >> num;

    cout << linear_search_num(arr, n, num);
}
```

Question: longest_consecutive_subsequence

```cpp
#include <bits/stdc++.h>
using namespace std;

// Linear search to check if a given element exists in the array
bool linearSearch(int arr[], int n, int x)
{
    for (int i = 0; i < n; i++)
    {
        if (arr[i] == x)
            return true;
    }
    return false;
}

/*
pair<int, vector<int>> find_lcs(int arr[], int n)
{
    int longest = 1;
    vector<int> longestSeq;  // This will store the longest subsequence

    for (int i = 0; i < n; i++)
    {
        int x = arr[i];
        int count = 1;
        vector<int> currentSeq = {x};  // This will store the current subsequence

        // Start looking for a consecutive subsequence
        while (linearSearch(arr, n, x + 1))
        {
            x = x + 1;
            count++;
            currentSeq.push_back(x);  // Add the next number to the subsequence
        }

        // If the current subsequence is longer, update the longest subsequence
        if (count > longest)
        {
            longest = count;
            longestSeq = currentSeq;
        }
    }

    return {longest, longestSeq};  // Return the length and the subsequence
}
*/

int find_lcs(int arr[], int n)
{
    int longest = 1;

    for (int i = 0; i < n; i++)
    {
        int x = arr[i];
        int count = 1;

        while (linearSearch(arr, n, x + 1))
        {
            x = x + 1;
            count++;
        }

        longest = max(longest, count);
    }

    return longest;
}

int find_lcs_better(vector<int> &nums)
{
    if (nums.size() == 0)
        return 0;
    sort(nums.begin(), nums.end());
    int n = nums.size();
    int lastSmaller = INT_MIN;
    int currentCount = 0;
    int longest = 1;
    for (int i = 0; i < n; i++)
    {
        if (nums[i] - 1 == lastSmaller)
        {
            currentCount += 1;
            lastSmaller = nums[i];
        }
        else if (lastSmaller != nums[i])
        {
            currentCount = 1;
            lastSmaller = nums[i];
        }
        longest = max(longest, currentCount);
    }
    return longest;
}

int find_lcs_optimal(vector<int> &a)
{
    int n = a.size();
    if (n == 0)
        return 0;
    int longest = 1;
    unordered_set<int> st;
    for (int i = 0; i < n; i++)
    {
        st.insert(a[i]);
    }

    for (auto it : st)
    {
        if (st.find(it - 1) == st.end())
        {
            int count = 1;
            int x = it;
            while (st.find(x + 1) != st.end())
            {
                x = x + 1;
                count += 1;
            }
            longest = max(longest, count);
        }
    }
    return longest;
}

int main()
{
    int arr[] = {1, 4, 3, 3, 2, 103, 100, 101, 101};
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Length of Longest Consecutive Subsequence is: " << find_lcs(arr, n) << endl;
    return 0;

    // vector<int> nums = {100, 4, 200, 1, 3, 2};

    // Find and output the length of the longest consecutive subsequence
    // cout << "Length of Longest Consecutive Subsequence is: " << find_lcs_better(nums) << endl;

    // return 0;
}
```

Question: longest_subarray_with_sum_k

```cpp
#include <bits/stdc++.h>
using namespace std;

// contiguous part of the array is called subarray
int find_longest_subArr(int arr[], int n, int num)
{
    // total tc : O(N^2)
    int length = 0;
    for (int i = 0; i < n; i++) // tc : O(N)
    {
        int sum = 0;
        for (int j = i; j < n; j++) // tc : O(N)
        {
            sum += arr[j];
            if (sum == num)
            {
                length = max(length, j - i + 1);
            }
        }
    }
    cout << "Length is : " << length << endl;
    return length;
}

int longestSubarrayWithSumKBetter(vector<int> arr, long long k)
{
    // tc : O(N * log N) N for iteration and log N for using ordered map(hashmap)
    // if unordered map is used then tc can be O(N * 1) but in worst can that 1 can become N when it has a lot of co
    // sc : O(N)
    map<long long, int> prefixSumMap;
    long long sum = 0;
    int maxLength = 0;
    for (int i = 0; i < arr.size(); i++)
    {
        sum += arr[i];
        if (sum == k)
        {
            maxLength = max(maxLength, i + 1);
        }

        long long remaining = sum - k;
        if (prefixSumMap.find(remaining) != prefixSumMap.end())
        {
            int length = i - prefixSumMap[remaining];
            maxLength = max(maxLength, length);
        }
        if (prefixSumMap.find(sum) == prefixSumMap.end())
        {
            prefixSumMap[sum] = i;
        }
    }
    return maxLength;
}

int longestSubarrayWithSumKOptimal(vector<int> arr, long long k)
{
    // study in detail self
    int left = 0, right = 0;
    long long sum = arr[0];
    int maxLength = 0;
    int n = arr.size();

    while (right < n)
    {
        while (left <= right && sum > k)
        {
            sum -= arr[left];
            left++;
        }

        if (sum == k)
        {
            maxLength = max(maxLength, right - left + 1);
        }
        right++;
        if (right < n)
            sum += arr[right];
    }
    return maxLength;
    // for(int i = 0; i<arr.size(); i++){
    //     for(int j = i; j<arr.size(); j++){
    //         sum += arr[j];
    //         if(sum == k) i++;
    //     }
    // }
}

int main()
{
    // int n;
    // cin >> n;
    // int arr[n];                 // sc : O(1)
    // for (int i = 0; i < n; i++) // tc : O(N)
    // {
    //     cin >> arr[i];
    // }

    // int num;
    // cin >> num;

    // cout << find_longest_subArr(arr, n, num) << endl;

    vector<int> arr;
    long long k;

    // --- Static input  ---
    /*
    arr = {1, 2, 3, 1, 1, 1, 1};
    k = 5;
    */

    // --- User input ---
    int n;
    cout << "Enter number of elements in the array: ";
    cin >> n;

    cout << "Enter " << n << " elements: ";
    arr.resize(n);
    for (int i = 0; i < n; ++i)
    {
        cin >> arr[i];
    }

    cout << "Enter the target sum k: ";
    cin >> k;

    int result = longestSubarrayWithSumKOptimal(arr, k);
    cout << "Length of the longest subarray with sum " << k << " is: " << result << endl;

    return 0;
}
```

Question: MajorityElement

```
// given an array, find element that appears more than n/2 times
// for example, array has 9 elements, so we will be looking for element which appears more than 4 times

#include <bits/stdc++.h>
using namespace std;

// brute force
void find_majority_element(int arr[], int n)
{
    // tc : O(N^2) since for loop inside for loop and both running till n
    // int vis[n] = {0};
    for (int i = 0; i < n; i++)
    {
        int count = 0;
        for (int j = 0; j < n; j++)
        {
            if (arr[i] == arr[j])
                count++;
        }
        if (count > n / 2)
        {
            cout << "Majority Element is : " << arr[i] << endl;
            return;
        }
    }
    cout << "No majority element found." << endl;
}

int find_majority_element_better(vector<int> v)
{
    map<int, int> mpp;
    // Time Complexity: O(N log N) for inserting into the map + O(N) for iterating through the map
    // Overall time complexity: O(N log N)
    // O(N log N) + O(N) which simplifies to O(NlogN)

    for (int i = 0; i < v.size(); i++)
    {
        mpp[v[i]]++;
    }

    for (auto it : mpp)
    {
        if (it.second > (v.size() / 2))
        {
            return it.first;
        }
    }
    return -1;
}

int find_majority_element_optimal(vector<int> v)
{
    // tc : O(N) + O(N). the second O(N) will only happen if there's a chance of not having a majority element, else
    // sc : O(1)
    int count = 0;
    int element;
    for (int i = 0; i < v.size(); i++)
    {
        if (count == 0)
        {
            count = 1;
            element = v[i];
        }

        else if (v[i] == element)
        {
            count++;
        }

        else
        {
            count--;
        }
    }
    int count1 = 0;
    for (int i = 0; i < v.size(); i++)
    {
        if (v[i] == element)
            count1++;
    }

    if (count1 > (v.size() / 2))
    {
        return element;
    }
    return -1;
}

int main()
{
    /*
    int n;
    cin >> n;
    int arr[n];                 // sc : O(1)
    for (int i = 0; i < n; i++) // tc : O(N)
    {
        cin >> arr[i];
    }

    find_majority_element(arr, n);
    */

    vector<int> arr;
    int n;

    cout << "Enter number of elements: ";
    cin >> n;
    /*
    arr = {2, 2, 1, 2, 2, 3, 2};
    n = arr.size();
    */

    // User input
    cout << "Enter " << n << " elements: ";
    for (int i = 0; i < n; i++)
    {
        int x;
        cin >> x;
        arr.push_back(x);
    }

    int majority = find_majority_element_optimal(arr);

    if (majority != -1)
        cout << "Majority element: " << majority << endl;
    else
        cout << "No majority element found." << endl;

    return 0;
}

--------------------------------------------------------------------------------

Question: majority_element_n_by_3

// a number should appear more than n/3 times in an array of n elements
// so in this question, we have to find the number of numbers that appear more than n/3 times. that is the count of

#include <bits/stdc++.h>
using namespace std;

list<int> find_maj_element(int nums[], int n)
{
    list<int> ls;
    for (int i = 0; i < n; i++)
    {
        // Problem with ls[0] != nums[i] check: Using ls[0], which is incorrect because std::list is not directly ir
        // if (ls.size() == 0 || ls[0] != nums[i])
        {
            int count = 0;
            for (int j = 0; j < n; j++)
            {
                if (nums[j] == nums[i])
                    count++;
                if (count > n / 3)
                    ls.push_back(nums[i]);
            }
        }
        if (ls.size() == 2)
            break;
    }
    return ls;
}

vector<int> find_majority_element(int nums[], int n)
{
    // tc : O(N ^ 2) and sc : O(1) coz ls will only store 2 elements
    vector<int> ls;

    // O(N)
    for (int i = 0; i < n; i++)
    {
        // Check if the number is already in the vector
        if (ls.size() == 0 || ls[0] != nums[i])
        {
            int count = 0;

            // Count the occurrences of nums[i]
            // O(N)
            for (int j = 0; j < n; j++)
            {
                if (nums[j] == nums[i])
                {
                    count++;
                }
            }

            if (count > n / 3)
            {
                ls.push_back(nums[i]);
            }
        }

        if (ls.size() == 2)
            break;
    }

    return ls;
}

// using hashmap
vector<int> find_majority_element_better(int arr[], int n)
{
    vector<int> ls;
    map<int, int> mpp;
    int mm = (n / 3) + 1;
    for (int i = 0; i < n; i++)
    {
        mpp[arr[i]]++;
        if (mpp[arr[i]] == mm)
        {
            ls.push_back(arr[i]);
        }
        if (ls.size() == 2)
            break;
    }

    return ls;
}

// his code for better approach using hashmap and sort the order of elements in answer
vector<int> majorityElement(vector<int> v)
{
    vector<int> ls;
    map<int, int> mpp;
    int n = v.size();
    int mini = (int)(n / 3) + 1;
    for (int i = 0; i < n; i++)
    {
        mpp[v[i]]++;
        if (mpp[v[i]] == mini)
        {
            ls.push_back(v[i]);
        }
        if (ls.size() == 2)
            break;
    }
    sort(ls.begin(), ls.end());
    return ls;
}

vector<int> majorityElementOptimal(vector<int> v)
{
    int count1 = 0, count2 = 0;
    int element1 = INT_MIN;
    int element2 = INT_MIN;
    for (int i = 0; i < v.size(); i++)
    {
        if (count1 == 0 && element2 != v[i])
        {
            count1 = 1;
            element1 = v[i];
        }
        else if (count2 == 0 && element1 != v[i])
        {
            count2 = 1;
            element2 = v[i];
        }

        else if (v[i] == element1)
```

```
            count1++;
        else if (v[i] == element2)
            count2++;

        else
        {
            count1--;
            count2--;
        }
    }
    vector<int> ls;
    count1 = 0, count2 = 0;
    for (int i = 0; i < v.size(); i++)
    {
        if (element1 == v[i])
            count1++;
        if (element2 == v[i])
            count2++;
    }

    int mini = (int)(v.size() / 3) + 1;
    if (count1 >= mini)
        ls.push_back(element1);
    if (count2 >= mini)
        ls.push_back(element2);
    sort(ls.begin(), ls.end());
    return ls;
}

int main()
{
    /*
    brute force approach

    int nums[] = {3, 1, 3, 3, 2, 2, 1, 1};
    int n = sizeof(nums) / sizeof(nums[0]);

    vector<int> majority_elements = find_majority_element(nums, n);

    for (int ele : majority_elements)
    {
        cout << ele << " ";
    }

    return 0;

    */

    // better approach using hashmap

    int arr1[] = {3, 1, 3, 3, 2, 2, 1, 1};
    int n1 = sizeof(arr1) / sizeof(arr1[0]);

    /*
    int n2;
    cout << "Enter number of elements: ";
    cin >> n2;

    int arr2[n2];
    cout << "Enter " << n2 << " elements: ";
    for (int i = 0; i < n2; i++) {
        cin >> arr2[i];
    }
    */

    vector<int> majority_elements = find_majority_element_better(arr1, n1);

    // vector<int> majority_elements = find_majority_element_better(arr2, n2);

    cout << "Majority elements (appears more than n/3 times): ";
    for (int ele : majority_elements)
    {
        cout << ele << " ";
    }
    cout << endl;

    return 0;
```

```
}
```

Question: MaxSubarraySumWithKadaneAlgo

```
#include <bits/stdc++.h>
using namespace std;

int maxSubarrayBrute(int arr[], int n, int num)
{
    // tc : O(N^3)
    int maxi = INT_MIN;
    for (int i = 0; i < n; i++)
    {
        for (int j = i; j < n; j++)
        {
            int sum = 0;
            for (int k = i; k <= j; k++)
            {
                sum += arr[k];
                maxi = max(sum, maxi);
            }
        }
    }
    return maxi;
}

int maxSubarrayBetter(int arr[], int n, int num)
{
    // tc : O(N^2)
    int maxi = INT_MIN;
    for (int i = 0; i < n; i++)
    {
        int sum = 0;
        for (int j = i; j < n; j++)
        {

            sum += arr[j];
            maxi = max(sum, maxi);
        }
    }
    return maxi;
}

long long maxSubarraySumOptimal(int arr[], int n)
{
    long long sum = 0;
    long long maxi = LONG_LONG_MIN;
    int start = 0, ansStart = -1, ansEnd = -1;
    for (int i = 0; i < n; i++)
    {
        if (sum == 0)
            start = i;
        sum += arr[i];

        if (sum > maxi)
        {
            maxi = sum;
            ansStart = start, ansEnd = i;
        }

        if (sum < 0)
        {
            sum = 0;
        }
    }
    cout << "Subarray with max sum: ";
    for (int i = ansStart; i <= ansEnd; i++)
        cout << arr[i] << " ";
    cout << endl;
    return maxi;
}

int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    cout << maxSubarraySumOptimal(arr, n) << endl;
    return 0;
}
```

Question: max_consecutive_ones

```
#include <bits/stdc++.h>
using namespace std;

int findMaxConsecutive(int arr[], int n)
{
    int maxi = 0;
    int count = 0;
    for (int i = 0; i < n; i++)
    {
        if (arr[i] == 1)
        {
            count++;
            maxi = max(maxi, count);
        }
        else
        {
            count = 0;
        }
    }
    return maxi;
}

// same using vector
int findMaxConseVec(vector<int> &nums)
{
    int maxi = 0;
    int count = 0;
    for (int i = 0; i < nums.size(); i++)
    {
        if (nums[i] == 1)
        {
            count++;
            maxi = max(maxi, count);
        }
        else
        {
            count = 0;
        }
    }
    return maxi;
}

int main()
{
    // int n;
    // cin >> n;
    // int arr[n];
    // for (int i = 0; i < n; i++)
    // {
    //     cin >> arr[i];
    // }

    // cout << findMaxConsecutive(arr, n) << endl;

    // Static input (Comment out if using dynamic input)
    /*
    vector<int> nums = {1, 1, 0, 1, 1, 1, 0, 1, 1};
    cout << "Maximum length of consecutive 1's: " << findMaxConseVec(nums) << endl;
    */

    // User input version
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;

    vector<int> nums(n);
    cout << "Enter the elements of the array (0's and 1's only): ";
    for (int i = 0; i < n; i++)
    {
        cin >> nums[i];
    }

    cout << "Maximum length of consecutive 1's: " << findMaxConseVec(nums) << endl;

    return 0;
}

--------------------------------------------------------------------------------

Question: max_product_subarray

#include <bits/stdc++.h>
using namespace std;

// brute force approach
int maxSubarrayProduct(int arr[], int n)
{
    // tc : ~ O(N^3)
    int maxi = INT_MIN;
    for (int i = 0; i < n; i++)
    {
        for (int j = i; j < n; j++)
        {
            int product = 1;
            for (int k = i; k <= j; k++)
            {
                product *= arr[k];
                maxi = max(product, maxi);
            }
        }
    }
    return maxi;
}

// better approach
int maxSubarrayProductBetter(vector<int> arr)
{
    // tc : O(N ^ 2)
    int n = arr.size();
    int maxi = INT_MIN;
    for (int i = 0; i < n; i++)
    {
        int product = 1;
        for (int j = i; j < n; j++)
        {
            product *= arr[j];
            maxi = max(maxi, product);
        }
    }
    return maxi;
}

// optimal approach (observation based, check how many are psoitive and negative in given array)
int maxSubarrayProductOptimal(vector<int> arr)
{
    // tc : O(N)
    int n = arr.size();
    int prefix = 1, suffix = 1;
    int maxi = INT_MIN;
    for (int i = 0; i < n; i++)
    {

        if (prefix == 0)
            prefix = 1;
        if (suffix == 0)
            suffix = 1;

        prefix = prefix * arr[i];         // from the start
        suffix = suffix * arr[n - i - 1]; // from the end
        maxi = max(maxi, max(prefix, suffix));
    }
    return maxi;
}

int main()
{
    int n;
    cin >> n;
    vector<int> arr(n);
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    cout << maxSubarrayProductOptimal(arr) << endl;
    return 0;
}

--------------------------------------------------------------------------------

Question: merge_overlapping_subIntervals

// in this problem, we are given a set of intervals and we need to merge all overlapping intervals
// for eg : (1,3) (2,6) (8,9) (9,11) (8,10) (2,4) (15,18) (16,17) will be merged to (1,6) (8,11) (15,18)

// brute force approach
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> mergeOverlappingIntervals(vector<vector<int>> &arr)
{
    int n = arr.size();
    sort(arr.begin(), arr.end()); // sort the intervals based on the starting point
    vector<vector<int>> ans;
    for (int i = 0; i < n; i++)
    {
        int start = arr[i][0];
        int end = arr[i][1];
        if (!ans.empty() && end <= ans.back()[1])
            continue; // if the current interval is completely inside the last interval in ans, skip it
        for (int j = i + 1; j < n; j++)
        {
            if (arr[j][0] <= end)
            {
                end = max(end, arr[j][1]); // merge the intervals
            }
            else
            {
                break; // no more overlapping intervals
            }
        }
        ans.push_back({start, end}); // add the merged interval to ans
    }
    return ans;
}

// better approach
vector<vector<int>> mergeOverlappingIntervalsBetter(vector<vector<int>> &arr)
{
    int n = arr.size();
    sort(arr.begin(), arr.end()); // sort the intervals based on the starting point
    vector<vector<int>> ans;
    for (int i = 0; i < n; i++)
    {
        if (ans.empty() || arr[i][0] > ans.back()[1])
        {                           // if current interval does not overlap with the last interval in ans}
            ans.push_back(arr[i]); // add the current interval to ans
        }
        else
        {
            ans.back()[1] = max(ans.back()[1], arr[i][1]); // merge the current interval with the last interval in a
        }
    }
    return ans;
}
```

Question: merge_sorted_arrays_without_extra_space

```cpp
#include <bits/stdc++.h>
using namespace std;

// brute force approach where tc : O(n+m)(for storing all elements in arr3) + O(n+m)(for merging the two arrays) = O
// sc : O(n+m) for storing the merged elements in arr3
void merge(long long arr1[], long long arr2[], int n, int m)
{
    long long arr3[n + m];
    int left = 0;
    int right = 0;
    int index = 0;

    // merge the two sorted arrays into arr3
    while (left < n && right < m)
    {
        // compare the elements of both arrays and add the smaller one to arr3
        // increment the index of the array from which the element was taken
        if (arr1[left] <= arr2[right])
        {
            arr3[index] = arr1[left];
            left++, index++;
        }
        else
        {
            // if the element in arr2 is smaller, add it to arr3
            // increment the index of arr2
            arr3[index] = arr2[right];
            right++, index++;
        }
    }

    // if there are remaining elements in arr1, add them to arr3
    while (left < n)
    {
        arr3[index] = arr1[left++];
    }

    // if there are remaining elements in arr2, add them to arr3
    while (right < m)
    {
        arr3[index] = arr2[right++];
    }

    for (int i = 0; i < n + m; i++)
    {
        if (i < n)
            arr1[i] = arr3[i]; // this line is important as it fills the first array with the merged elements
        else
            arr2[i - n] = arr3[i]; // this line is important as it fills the second array with the remaining element
    }
}

// optiml approach 1 where tc : O(min(n+m)) + O(n log n) + O(m log m) = O(nlogn + mlogm)
// sc : O(1) as we are not using any extra space
void mergeOptimal1(long long arr1[], long long arr2[], int n, int m)
{
    int left = n - 1; // last index of arr1
    int right = 0;    // first index of arr2

    // we will swap elements from arr1 and arr2 until we find the correct position for the elements in arr1
    // we will do this by comparing the last element of arr1 with the first element of arr2
    // if the last element of arr1 is greater than the first element of arr2,
    // we will swap them and move the pointers accordingly
    // we will continue this process until we reach the end of arr1 or the end of arr2
    // this way we will ensure that both arrays are sorted and we will not use any extra space
    // this is a two pointer approach
    while (left >= 0 && right < m)
    {
        if (arr1[left] > arr2[right])
        {
            swap(arr1[left], arr2[right]);
            left--, right++;
        }
        else
        {
            break; // if the current element in arr1 is less than or equal to the current element in arr2, we can st
        }
    }

    sort(arr1, arr1 + n); // sort arr1
    sort(arr2, arr2 + m); // sort arr2
}

void swapIfGreater(long long arr1[], long long arr2[], int index1, int index2)
{
    if (arr1[index1] > arr2[index2])
    {
        swap(arr1[index1], arr2[index2]);
    }
}

// optimal approach 2, using shell sort (gap method)
// where tc : O((n+m) * log2(n+m)) and sc : O(1) as we are not using any extra space
// this approach is based on the idea of shell sort, where we will compare elements at a certain gap and swap them i
// we will keep reducing the gap until it becomes 0, and at that point, both
// arrays will be sorted
// this is a two pointer approach
void mergeOptimal2(long long arr1[], long long arr2[], int n, int m)
{
    int length = n + m;                    // total length of both arrays
    int gap = (length / 2) + (length % 2); // calculate the gap, len /2 will give us the half of the length, and if
    while (gap > 0)
    {
        int left = 0;           // pointer for arr1
        int right = left + gap; // pointer for arr2
        while (right < length)
        {
            // comparing elements of arr1 and arr2
            if (left < n && right >= n)
            {
                // here right - n is used to get the index of arr2
                // as right is the index of the merged array, we need to subtract n to get
                // the index of arr2
                // if left is in arr1 and right is in arr2
                // we will compare the elements of arr1 and arr2
                // if the element in arr1 is greater than the element in arr2, we will
                // swap them
                // this is done to ensure that both arrays are sorted
                swapIfGreater(arr1, arr2, left, right - n); // swap if the element in arr1 is greater than the eleme
            }

            // comparing elements of arr2 and arr2 (both  pointer are in arr2)
            else if (left >= n)
            {
                swapIfGreater(arr2, arr2, left - n, right - n); // here left - n and right - n are used to get the i
            }

            // comparing elements of arr1 and arr1 (both pointer are in arr1)
            else
            {
                swapIfGreater(arr1, arr2, left, right); // here right is the index of arr1, so we can directly use i
            }
            left++, right++; // increment both pointers to move to the next elements in the arrays
        }

        if (gap == 1)
            break;                   // if gap is 1, we can stop as we have already sorted the arrays
        gap = (gap / 2) + (gap % 2); // calculate the new gap, len /2 will give us the half of the length, and if th
    }
}

--------------------------------------------------------------------------------

Question: next_permutation

#include <bits/stdc++.h>
using namespace std;

// dictionary wise arrangement
```

```
vector<int> nextGreaterPermutation(vector<int> &a)
{
    int index = -1;
    // if(index == -1) reverse(A);
    int n = a.size();
    for (int i = n - 2; i >= 0; i--)
    {
        if (a[i] < a[i + 1])
        {
            index = i;
            break;
        }
    }

    if (index == -1)
    {
        reverse(a.begin(), a.end());
        return a;
    }

    for (int i = n - 1; i >= index; i--)
    {
        if (a[i] > a[index])
        {
            swap(a[i], a[index]);
            break;
        }
    }

    reverse(a.begin() + index + 1, a.end());
    return a;
}

int main()
{
    vector<int> a = {1, 5, 3, 4, 9, 6, 7, 2};
    vector<int> result = nextGreaterPermutation(a);

    cout << "Next greater permutation: ";
    for (int i = 0; i < result.size(); i++)
    {
        cout << result[i] << " ";
    }
    cout << endl;
    return 0;
}

/*

### Input Vector:

`a = {1, 5, 3, 4, 9, 6, 7, 2}`

### Step-by-Step Process:

#### **Step 1: Find the rightmost pair where `a[i] < a[i+1]`.**

Start from the second-to-last element (`a[n-2]`) and look for the first position `i` where `a[i] < a[i + 1]`. This i

* Compare `a[6] (7)` and `a[7] (2)`: Since `7 > 2`, continue.
* Compare `a[5] (6)` and `a[6] (7)`: Since `6 < 7`, this is the **first pair** where `a[i] < a[i + 1]`. We have foun

#### **Step 2: Find the largest element `a[j]` such that `a[j] > a[i]` and `j > i`.**

Now that we've found `i = 5` where `a[i] < a[i + 1]`, we need to find the largest element that is greater than `a[i]

* Start from the end of the array and look for a number greater than `a[5] (6)`.
* Compare `a[7] (2)` with `a[5] (6)`: Since `2 < 6`, continue.
* Compare `a[6] (7)` with `a[5] (6)`: Since `7 > 6`, we choose `j = 6` (value `7`).

#### **Step 3: Swap `a[i]` and `a[j]`.**

Now that we have found `i = 5` and `j = 6`, we swap `a[5]` with `a[6]`.

Before the swap:

* `a = {1, 5, 3, 4, 9, 6, 7, 2}`
```

```
After the swap:

* `a = {1, 5, 3, 4, 9, 7, 6, 2}`

#### **Step 4: Reverse the subarray from `i+1` to the end.**

Finally, to ensure that the next permutation is the smallest lexicographically larger permutation, we reverse the po

* The subarray starting from `i + 1 = 6` is `{6, 2}`.
* Reverse the subarray `{6, 2}` to get `{2, 6}`.

After reversing:

* `a = {1, 5, 3, 4, 9, 7, 2, 6}`

### Final Result:

The next greater permutation is:
`{1, 5, 3, 4, 9, 7, 2, 6}`

### Summary of the Steps:

1. **Find the first pair `a[i] < a[i+1]`**: We found `i = 5`.
2. **Find the largest `a[j] > a[i]` for `j > i`**: We found `j = 6`.
3. **Swap `a[i]` and `a[j]`**: The array becomes `{1, 5, 3, 4, 9, 7, 6, 2}`.
4. **Reverse the subarray from `i + 1` to the end**: The array becomes `{1, 5, 3, 4, 9, 7, 2, 6}`.

This gives us the next lexicographically greater permutation.


*/

--------------------------------------------------------------------------------

Question: num_of_subarrays_with_xor_k

#include <bits/stdc++.h>
using namespace std;

// answer to be verified from this, doesn't give all subarrays
// https://www.geeksforgeeks.org/count-number-subarrays-xor-k/
int find_num_subarray_xor_k(vector<int> &arr, int k)
{
    int n = arr.size();
    int count = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = i; j < n; j++)
        {
            int xor_sum = 0;
            for (int l = i; l < j; l++)
            {
                xor_sum = xor_sum ^ arr[l];
            }
            if (xor_sum == k)
                count++;
        }
    }
    return count;
}

// better approach
int find_num_subarray_xor_k_better(vector<int> &arr, int k)
{
    int n = arr.size();
    int count = 0;
    for (int i = 0; i < n; i++)
    {
        int xor_sum = 0;
        for (int j = i; j < n; j++)
        {
            xor_sum = xor_sum ^ arr[j];
            if (xor_sum == k)
                count++;
        }
    }
    return count;
}

// optimal approach using prefix XOR and hash map
int subarraysWithXOR_K(vector<int> a, int k)
{
    int xr = 0;
    map<int, int> mpp;
    mpp[xr]++; // Initialize with 0 XOR count i.e., {0,1}
    int count = 0;
    for (int i = 0; i < a.size(); i++)
    {
        xr = xr ^ a[i];

        // now that we have xr, we are looking for (xr ^ k) in the map
        int x = xr ^ k;
        // if x is present in the map, it means we have found some subarrays with
        // XOR equal to k
        count += mpp[x];
        mpp[xr]++; // Increment the count of current XOR in the map
    }
    return count;
}

int main()
{
    int n, k;
    cout << "Enter number of elements in the array: ";
    cin >> n;

    vector<int> arr(n);
    cout << "Enter the elements of the array:\n";
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    cout << "Enter the value of k: ";
    cin >> k;

    int result = find_num_subarray_xor_k(arr, k);
    cout << "Number of subarrays with XOR equal to " << k << " is: " << result << endl;

    return 0;
}
```

Question: one_left_rotate

```
#include <bits/stdc++.h>
using namespace std;

void left_rot(int arr[], int n)
{
    // t.c. is O(N) coz for loop running till n-1, and s.c. is O(1) bcoz all operations are happening in given array
    int temp = arr[0];
    for (int i = 0; i < n - 1; i++) // here, nos. are swapped only till n-1, bcoz nth element is swapped with first
    {
        arr[i] = arr[i + 1];
    }
    arr[n - 1] = temp;
    // return arr[n];
}

void print_array(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    left_rot(arr, n);
    print_array(arr, n);
    return 0;
}
```

Question: only_one_no_appear_once

```
#include <bits/stdc++.h>
using namespace std;

// my code, fell short, incorrect
int find_no_app_once(int arr[], int n)
{
    int xor1 = 0;
    int xor2 = 0;
    for (int i = 0; i < n; i++)
    {
        xor2 = xor2 ^ arr[i];
        xor1 = xor1 ^ (i + 1);
    }
    // xor1 = xor1 ^ n;
    return xor1 ^ xor2;
}

// optimal approach using xor
int find_no_app_once_optimal(int arr[], int n)
{
    // tc : O(N) & sc : O(1)
    int xor1 = 0;
    for (int i = 0; i < n; i++)
    {
        xor1 = xor1 ^ arr[i];
    }
    return xor1;
}

// brute force by linear search method
int find_no_app_once_brute(int arr[], int n)
{
    for (int i = 0; i < n; i++) // tc: O(N^2) coz nested for loop both till n & sc : 0(1)
    {
        int num = arr[i];
        int count = 0;
        for (int j = 0; j < n; j++)
        {
            if (arr[j] == num)
                count++;
        }
        if (count == 1)
            return num;
    }
    return -1;
}

int find_no_app_once_better(int arr[], int n)
{
    int maxi = arr[0];
    for (int i = 0; i < n; i++)
    {
        maxi = max(maxi, arr[i]);
    }
    int hash[maxi] = {0};
    for (int i = 0; i < n; i++)
    {
        hash[arr[i]]++;
    }
    for (int i = 0; i < n; i++)
    {
        if (hash[arr[i]] == 1)
            return arr[i];
    }
    return -1;
}

int find_no_app_once_better_map(int arr[], int n)
{
    map<int, int> mpp1;
    for (int i = 0; i < n; i++)
    {
        mpp1[arr[i]]++;
    }

    for (auto it : mpp1)
    {
        if (it.second == 1)
        {
            return it.first;
        }
    }
    return -1;

    // total tc : O(N log M) + O(N/2 + 1) where N is size of array and M is size of map which is (N/2 + 1)
    // sc : O(N/2 + 1)
}

int main()
{
    int n;
    cin >> n;
    int arr[n];                 // sc : O(1)
    for (int i = 0; i < n; i++) // tc : O(N)
    {
        cin >> arr[i];
    }

    cout << find_no_app_once_optimal(arr, n) << endl;
}
```

Question: pascal_triangle

```
/*
There can be 3 types of question in this :

1. given row and col., find element at that place, eg : r = 5, c = 3

2. print the n^th row of pascal triangle, eg : n = 5 -> 1 4 6 4 1

3. given n. print the entire pascal triangle till n^th row.
*/

#include <bits/stdc++.h>
using namespace std;

// find element at given row and column
int nCr(int n, int r)
{
    // tc : O(r) and sc : O(1)
    long long resultant = 1;
    for (int i = 0; i < r; i++)
    {
        resultant = resultant * (n - i);
        resultant = resultant / (i + 1);
    }
    return resultant;
}

// second type : print n^th row
void nCrFullRow(int n)
{ // tc : O(n * r)
    // here since pascal triangle is one based index, that is it starts from 1, so we have taken r from 1 and condit
    // if it was zero based index, then we would have taken r = 0, and condition : (n,r)
    for (int r = 1; r <= n; r++)
    {
        if (r > 1)
            cout << " "; // this is for space b/w nos.
        cout << nCr(n - 1, r - 1);
    }
    cout << endl;
}

// 3rd type : print all elements till n^th row
list<list<int>> nCrTillNthRow(int n)
{
    // here tc : approx to O(n * n * r) which can be near to n^3 at worst
    list<list<int>> ans;
    for (int r = 1; r <= n; r++)
    {
        list<int> tempList;
        for (int c = 1; c <= r; c++)
        {
            tempList.push_back(nCr(r - 1, c - 1));
        }
        ans.push_back(tempList);
    }
    return ans;
}

// optimal approach for third type
vector<int> generateRow(int row)
{
    long long ans = 1;
    vector<int> ansRow;
    ansRow.push_back(1);
    for (int col = 1; col < row; col++)
    {
        ans = ans * (row - col);
        ans = ans / (col);
        ansRow.push_back(ans);
    }
    return ansRow;
}

vector<vector<int>> pascalTriangleFullTillN(int n)
{
    vector<vector<int>> ans;
    for (int i = 1; i <= n; i++)
    {
        ans.push_back(generateRow(i));
    }
    return ans;
}

int main()
{
    int n, r;
    cin >> n;
    // cin >> r;
    // cout << nCr(n, r) << endl;

    // second type
    // nCrFullRow(n);

    // third type
    // list<list<int>> pascalTriangle = nCrTillNthRow(n);

    // print pascal traingle's rows
    // for (auto &row : pascalTriangle)
    // {
    //     for (auto &num : row)
    //     {
    //         cout << num << " ";
    //     }
    //     cout << endl;
    // }

    // third type optimal way
    vector<vector<int>> pascalTriangle = pascalTriangleFullTillN(n);

    for (const auto &row : pascalTriangle)
    {
        for (const auto &num : row)
        {
            cout << num << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Question: rearrange_positive_negative_alternately

```
#include <bits/stdc++.h>
using namespace std;

// arrange negative and positive elements alternately
int rearrange_elements(int arr[], int n)
{
    int pos[n];
    int neg[n];
    int pos_count = 0, neg_count = 0;
    for (int i = 0; i < n; i++)
    {
        if (arr[i] > 0)
        {
            pos[pos_count++] = arr[i];
        }
        else
        {
            neg[neg_count++] = arr[i];
        }
    }

    for (int i = 0; i < n / 2; i++)
    {
        arr[2 * i] = pos[i];
        arr[2 * i + 1] = neg[i];
    }
    // return arr[n];
}

// vector approach, his code
vector<int> rearrangeArray(vector<int> &nums)
{
    int n = nums.size();
    vector<int> ans(n, 0);
    int posIndex = 0, negIndex = 1;
    for (int i = 0; i < n; i++)
    {
        if (nums[i] < 0)
        {
            ans[negIndex] = nums[i];
            negIndex += 2;
        }
        else
        {
            ans[posIndex] = nums[i];
            posIndex += 2;
        }
    }
    return ans;
}

vector<int> rearrangeArrayPosNotEqualToNeg(vector<int> &a)
{
    vector<int> pos, neg;
    int n = a.size();
    for (int i = 0; i < n; i++)
    {
        if (a[i] > 0)
        {
            pos.push_back(a[i]);
        }
        else
        {
            neg.push_back(a[i]);
        }
    }
    if (pos.size() > neg.size())
    {
        for (int i = 0; i < neg.size(); i++)
        {
            a[2 * i] = pos[i];
            a[2 * i + 1] = neg[i];
        }
        int index = neg.size() * 2;
        for (int i = neg.size(); i < pos.size(); i++)
        {
            a[index] = pos[i];
            index++;
        }
    }

    else
    {
        for (int i = 0; i < pos.size(); i++)
        {
            a[2 * i] = pos[i];
            a[2 * i + 1] = neg[i];
        }
        int index = pos.size() * 2;
        for (int i = pos.size(); i < pos.size(); i++)
        {
            a[index] = neg[i];
            index++;
        }
    }
    return a;
}

int main()
{
    /*
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    if (rearrange_elements(arr, n) == 0)
    {
        // Output the rearranged array
        for (int i = 0; i < n; i++)
        {
            cout << arr[i] << " ";
        }
        cout << endl;
    }

    */

    // vector<int> nums = {1, 5, 3, -4, -9, -6};
    vector<int> a = {1, 5, 3, -4, -9, -6, 7, 2};

    // Dynamic input for array size and elements
    // int n;
    // cin >> n;

    // // Check if n is even
    // if (n % 2 != 0)
    // {
    //     cout << "Array size must be even!" << endl;
    //     return -1;
    // }

    // vector<int> nums(n);

    // // Input elements
    // for (int i = 0; i < n; i++)
    // {
    //     cin >> nums[i];
    // }

    // Get the rearranged array
    // vector<int> result = rearrangeArray(nums);
    vector<int> result = rearrangeArrayPosNotEqualToNeg(a);

    // Output the rearranged array
    for (int i = 0; i < a.size(); i++)
    {
        cout << result[i] << " ";
    }
    cout << endl;
    return 0;
}

--------------------------------------------------------------------------------

Question: remove_duplicates

#include <bits/stdc++.h>
using namespace std;

int remove_dup()
{

    /*

    Total Time Complexity:

    O(N) for input reading

    O(N * log N) for sorting the vector

    O(N * log N) for inserting elements into the set

    O(N) for copying the unique elements back into the vector and printing them

    O(1) for returning the size of the set

    So, the total time complexity is:

    O(N * log N) + O(N * log N) + O(N) = O(N * log N)

    */

    int n;
    cin >> n; // Input size of the array

    vector<int> a(n); // O(1) for creating an empty vector, O(N) for allocation of the vector
    for (int i = 0; i < n; i++)
    {
        cin >> a[i]; // O(N) for reading the input
    }

    sort(a.begin(), a.end()); // O(N * log N) for sorting the array

    set<int> st; // O(1) for creating an empty set

    for (int i = 0; i < n; i++)
    {
        st.insert(a[i]); // O(log N) for each insertion into the set, O(N * log N) for N insertions
    }

    int index = 0;
    for (auto it : st)
    {
        a[index] = it;     // O(1) for each assignment
        index++;           // O(1)
        cout << it << " "; // O(1) for each print operation
    }
    cout << endl; // O(1) for newline output

    return st.size(); // O(1) for returning the size of the set
}

// Pseudocode for optimal remove duplicates from array
/*
i = 0;
for(j = 1; j<n; j++){
    if(arr[j] != arr[i]){
```

```
    arr[i+1] = arr[j];
    i++;
}
}
return (i+1);
*/

int removeDuplicatesOptimal(vector<int> &arr, int n)
{
    int i = 0;
    for (int j = 1; j < n; j++)
    {
        if (arr[i] != arr[j])
        {
            arr[i + 1] = arr[j];
            i++;
        }
    }
    return i + 1;
}

int main()
{
    // cout << remove_dup() << endl; // O(N * log N)
    // return 0;                     // O(1)

    // hardcoded array
    // vector<int> arr = {1, 1, 2, 2, 3, 4, 5, 7};
    // int n = arr.size();
    // int newLength = removeDuplicatesOptimal(arr, n);
    // cout << newLength << endl;

    int n;
    cout << "Enter the number of elements in the array: ";
    cin >> n;

    vector<int> arr(n);
    cout << "Enter the elements of the sorted array (space separated): ";
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    // Call the removeDuplicates function
    int newLength = removeDuplicatesOptimal(arr, n);

    // Output the result
    cout << "The new length after removing duplicates is: " << newLength << endl;
    cout << "The modified array is: ";
    for (int i = 0; i < newLength; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;
}


--------------------------------------------------------------------------------

Question: reverse_pairs

#include <bits/stdc++.h>
using namespace std;

// brute force approach, tc : O(N^2)
int count_reverse_pairs(vector<int> arr)
{
    int n = arr.size();
    int count_brute = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (arr[i] > 2 * arr[j])
                count_brute++;
        }
    }
    return count_brute;
}

// better approach (similar to count pairs(count inversion in array), using merge sort)

int main()
{
    vector<int> arr = {40, 25, 19, 9, 6, 2};
    int result = count_reverse_pairs(arr);
    cout << result << endl;
}
```

Question: rotate_matrix_by_90

```
#include <bits/stdc++.h>
using namespace std;

vector<vector<int>> rotateMatrix(vector<vector<int>> &matrix)
{
    int n = matrix.size();
    // vector<vector<int>> ans[n][n]; wrong syntax
    vector<vector<int>> ans(n, vector<int>(n));

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            ans[j][n - 1 - i] = matrix[i][j];
        }
    }
    return ans;
}

// optimal approach using transpose of matrix
void rotateMatrixOptimal(vector<vector<int>> &mat)
{
    int n = mat.size();
    // O(N/2 * N/2)
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            swap(mat[i][j], mat[j][i]);
        }
    }

    // reverse
    // O(N * N/2)
    for (int i = 0; i < n; i++)
    {
        // row is mat[i]
        reverse(mat[i].begin(), mat[i].end());
    }
}

int main()
{
    // brute force

    // vector<vector<int>> mat = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    // vector<vector<int>> rotated = rotateMatrix(mat);

    // for(auto row : rotated) {
    //     for(auto val : row) cout << val << " ";
    //     cout << endl;
    // }

    // return 0;

    // Optimal one
    // Example matrix
    vector<vector<int>> mat = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}};
    cout << "Original matrix:" << endl;
    for (auto row : mat)
    {
        for (auto val : row)
        {
            cout << val << " ";
        }
        cout << endl;
    }

    // Rotate the matrix
    rotateMatrixOptimal(mat);

    cout << "\nRotated matrix (90 degrees clockwise):" << endl;
    for (auto row : mat)
    {
        for (auto val : row)
        {
            cout << val << " ";
        }
        cout << endl;
    }

    return 0;
}

/*

for (int i = 0; i < n - 1; i++) {            // Outer loop for rows
    // The outer loop (`i`) is used to iterate through each row, except for the last one.
    // We don't need to loop over the last row because the transpose operation involves swapping
    // elements above the diagonal (i.e., elements where the row index is less than the column index).
    // By going from `i = 0` to `i < n - 1`, we are ensuring we don't try to swap elements along the diagonal
    // or swap elements that have already been processed. For example, we don't want to swap `mat[2][2]` with itself

    for (int j = i + 1; j < n; j++) {       // Inner loop for columns (only above the diagonal)
        // The inner loop (`j`) starts from `i + 1` to avoid swapping along the diagonal (where i == j),
        // and to only swap elements that are **above** the diagonal.
        // `j = i + 1` means that for each row `i`, the column `j` starts from the element just to the right
        // of the diagonal element (i.e., the element at `mat[i][i+1]`).

        // The condition `j < n` ensures that the column `j` stays within the bounds of the matrix.
        // This will only loop through the columns that are positioned **right** of the diagonal.

        // The swap operation then occurs between `mat[i][j]` and `mat[j][i]`,
        // which will transpose the matrix (swap rows and columns) above the diagonal.
        swap(mat[i][j], mat[j][i]);           // Swap element at (i, j) with element at (j, i)
    }
}


*/

--------------------------------------------------------------------------------

Question: second_largest

#include <bits/stdc++.h>
#include <climits>
using namespace std;

int sec_large_brute()
{
    // t.c. here is O(N log N + N) bcoz N log N for sorting and +N for traversing the whole array for finding sec_la

    int n;
    cin >> n;
    int arr[n];

    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    // O(N log N): Sorting the array in ascending order
    sort(arr, arr + n);
    // O(1): Accessing the largest element (last element after sorting)
    int largest = arr[n - 1];
    // O(1): Initializing second largest to an invalid value
    int sec_largest = -1;

    // O(N): Traversing the array from the second last element to find the second largest distinct element
    for (int i = n - 2; i >= 0; i--)
    {
        if (arr[i] != largest)
        {
            sec_largest = arr[i];
            break;
        }
    }
    // O(1): Printing the largest element
    cout << largest << endl;
    // O(1): Returning the second largest element
    return sec_largest;
}

int sec_lar_better()
{
    // here t.c is O(N + N) = O(2N) because:
    // - For finding largest, it is O(N)
    // - For finding sec_largest, it is O(N)
    // So the overall time complexity is O(N)
    int n;
    cin >> n;
    int arr[n];
    // O(N): Reading the array input
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    // O(N): Finding the largest element by traversing the array once
    int largest = arr[0];
    for (int i = 0; i < n; i++)
    {
        if (arr[i] > largest)
            largest = arr[i];
    }
    int sec_largest = -1;
    // O(N): Traversing the array again to find the second largest element
    for (int i = 0; i < n; i++)
    {
        if (arr[i] > sec_largest && arr[i] != largest)
        {
            sec_largest = arr[i];
        }
    }
    cout << "Largest value is: " << largest << endl;
    return sec_largest;
}

int secondLargest(vector<int> &arr, int n)
{
    // Time complexity: O(N), as we loop through the array once to find the second largest element.
    int largest = arr[0]; // O(1) initialization of largest value.
    int slargest = -1;    // O(1) initialization of second largest value.

    // Time complexity: O(N) for the loop, as it checks each element once.
    for (int i = 1; i < n; i++) // O(N) loop
    {
        if (arr[i] > largest) // O(1) comparison
        {
            slargest = largest; // O(1) assignment
            largest = arr[i];   // O(1) assignment
        }
        else if (arr[i] < largest && arr[i] > slargest) // O(1) comparison
        {
            slargest = arr[i]; // O(1) assignment
        }
    }
    return slargest; // O(1) return
}
```

```
int secondSmallest(vector<int> &arr, int n)
{
    // Time complexity: O(N), as we loop through the array once to find the second smallest element.
    int smallest = arr[0];   // O(1) initialization of smallest value.
    int ssmallest = INT_MAX; // O(1) initialization of second smallest value.

    // Time complexity: O(N) for the loop, as it checks each element once.
    for (int i = 1; i < n; i++) // O(N) loop
    {
        if (arr[i] < smallest) // O(1) comparison
        {
            ssmallest = smallest; // O(1) assignment
            smallest = arr[i];    // O(1) assignment
        }
        else if (arr[i] != smallest && arr[i] < ssmallest) // O(1) comparison
        {
            ssmallest = arr[i]; // O(1) assignment
        }
    }
    return ssmallest; // O(1) return
}

vector<int> optimalSecLarSecSmall(int n, vector<int> arr)
{
    // Time complexity: O(N) for finding second largest element.
    int slargest = secondLargest(arr, n); // O(N)

    // Time complexity: O(N) for finding second smallest element.
    int ssmallest = secondSmallest(arr, n); // O(N)

    // Space complexity: O(1) for creating a result vector and returning it.
    return {slargest, ssmallest}; // O(1) return
}

int main()
{
    int n;    // O(1) initialization of the variable n.
    cin >> n; // O(1) input for the size of the array.

    // Space complexity: O(N) for storing the array of size n.
    vector<int> arr(n); // O(N) space allocation for the vector.

    // Time complexity: O(N) for reading the array elements.
    for (int i = 0; i < n; i++) // O(N) loop
    {
        cin >> arr[i]; // O(1) input for each element.
    }

    // Time complexity: O(N) for calling optimalSecLarSecSmall, which internally calls two O(N) functions.
    vector<int> result = optimalSecLarSecSmall(n, arr); // O(N) due to secondLargest and secondSmallest.

    // Time complexity: O(1) for printing the second largest and second smallest elements.
    cout << "Second Largest: " << result[0] << endl;  // O(1)
    cout << "Second Smallest: " << result[1] << endl; // O(1)

    return 0; // O(1) return statement.
}
```

Question: set_matrix_zeros

```
#include <bits/stdc++.h>
using namespace std;

// Function to mark a row as -1 where there is a 0
void markRow(int arr[][100], int i, int m)
{
    for (int j = 0; j < m; j++)
    {
        if (arr[i][j] != 0)
        {
            arr[i][j] = -1;
        }
    }
}
```

```
// Function to mark a column as -1 where there is a 0
void markCol(int arr[][100], int j, int n)
{
    for (int i = 0; i < n; i++)
    {
        if (arr[i][j] != 0)
        {
            arr[i][j] = -1;
        }
    }
}

// Function to set rows and columns to -1 where there are 0s
void set_mat_zero(int arr[][100], int n, int m)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (arr[i][j] == 0)
            {
                markRow(arr, i, m);
                markCol(arr, j, n);
            }
        }
    }
}

// Function to convert all -1s to 0s after processing the matrix
void setFinalZero(int arr[][100], int n, int m)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (arr[i][j] == -1)
            {
                arr[i][j] = 0; // Convert the temporary -1 to 0
            }
        }
    }
}

// Function to print the matrix
void printMatrix(int arr[][100], int n, int m)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
}

// better solution with tc : O(2 * n * m) approx to O(2 N^2)
// sc : (n + m)
vector<vector<int>> zeroMatrix(vector<vector<int>> &matrix, int n, int m)
{
    int col[m] = {0};
    int row[n] = {0};
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (matrix[i][j] == 0)
            {
                row[i] = 1;
                col[j] = 1;
            }
        }
    }
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (row[i] || col[j])
            {
                matrix[i][j] = 0;
            }
        }
    }
    return matrix;
}

vector<vector<int>> zeroMatrixOptimal(vector<vector<int>> &matrix, int n, int m)
{
    // int col[m] = {0} -> matrix[0][..]
    // int row[n] = {0} -> matrix[..][0]
    int col0 = 1;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            if (matrix[i][j] == 0)
            {
                // mark the i-th row
                matrix[i][0] = 0;
                // mark the j-th column
                if (j != 0)
                    matrix[0][j] = 0;
                else
                    col0 = 0;
            }
        }
    }
    for (int i = 1; i < n; i++)
    {
        for (int j = 1; j < m; j++)
        {
            if (matrix[i][j] != 0)
            {
                // check for col & row
                if (matrix[0][j] == 0 || matrix[i][0] == 0)
                {
                    matrix[i][j] = 0;
                }
            }
        }
    }
    if (matrix[0][0] == 0)
    {
        for (int j = 0; j < m; j++)
            matrix[0][j] = 0;
    }

    if (col0 == 0)
    {
        for (int i = 0; i < n; i++)
        {
            matrix[i][0] = 0;
        }
    }
    return matrix;
}

int main()
{
    /*
    // brute force approach
    int arr[3][100] = {
        {1, 1, 0},
        {1, 0, 1},
        {1, 1, 1}};

    int n = 3, m = 3;

    cout << "Original Matrix: " << endl;
    printMatrix(arr, n, m);

    // Call the function to modify the matrix
    set_mat_zero(arr, n, m);
    // Convert all -1s to 0s
    setFinalZero(arr, n, m);

    cout << "\nModified Matrix: " << endl;
    printMatrix(arr, n, m);

    return 0;
    */

    // Static input
    vector<vector<int>> matrix = {
        {1, 1, 0},
        {1, 0, 1},
        {1, 1, 1}};
    int n = 3, m = 3;

    // Uncomment for user input
    /*
    // User input (commented out for now)
    int n, m;
    cout << "Enter number of rows and columns: ";
    cin >> n >> m;
    vector<vector<int>> matrix(n, vector<int>(m));

    cout << "Enter the matrix values: \n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            cin >> matrix[i][j];
        }
    }
    */

    cout << "Original Matrix: \n";
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }

    // Call zeroMatrix function
    matrix = zeroMatrix(matrix, n, m);

    cout << "\nModified Matrix: \n";
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Question: sort_arrays_of_012

```
#include <bits/stdc++.h>
using namespace std;

void sort_arr_012(int arr[], int n)
{
    // int count0 = 0, count1 = 1, count2 = 2; striver
    int count0 = 0, count1 = 0, count2 = 0; // correct
    for (int i = 0; i < n; i++)
    {
        if (arr[i] == 0)
            count0++;
        else if (arr[i] == 1)
            count1++;
        else
```

```
            count2++;
    }
    for (int i = 0; i < count0; i++)
        arr[i] = 0;
    for (int i = count0; i < count0 + count1; i++)
        arr[i] = 1;
    for (int i = count0 + count1; i < n; i++)
        arr[i] = 2;
}

// optimal approach using dutch flag(3 pointers)
void sortArray(vector<int> &arr, int n)
{
    int low = 0, mid = 0, high = n - 1;
    while (mid <= high)
    {
        if (arr[mid] == 0)
        {
            swap(arr[low], arr[mid]);
            low++;
            mid++;
        }
        else if (arr[mid] == 1)
        {
            mid++;
        }
        else
        {
            swap(arr[mid], arr[high]);
            high--;
        }
    }
}

int main()
{
    int n;
    cin >> n;
    // int arr[n];
    vector<int> arr(n);         // sc : O(1)
    for (int i = 0; i < n; i++) // tc : O(N)
    {
        cin >> arr[i];
    }

    sortArray(arr, n);
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Question: spiral_traversal_matrix

```
#include <bits/stdc++.h>
using namespace std;

vector<int> spiralMatrix(vector<vector<int>> &mat)
{
    int n = mat.size();
    int m = mat[0].size();
    int left = 0, right = m - 1;
    int top = 0, bottom = n - 1;
    vector<int> ans;

    while (top <= bottom && left <= right)
    {

        // right
        for (int i = left; i <= right; i++)
        {
            ans.push_back(mat[top][i]);
        }
        top++;

        for (int i = top; i <= bottom; i++)
        {
            ans.push_back(mat[i][right]);
        }
        right--;

        if (top <= bottom)
        {
            for (int i = right; i >= left; i--)
            {
                ans.push_back(mat[bottom][i]);
            }
            bottom--;
        }

        if (left <= right)
        {
            for (int i = bottom; i >= top; i--)
            {
                ans.push_back(mat[i][left]);
            }
            left++;
        }
    }
    return ans;
}
```

Question: tempCodeRunnerFile

```
#include <bits/stdc++.h>
using namespace std;
```

Question: to_right

```
#include <bits/stdc++.h>
using namespace std;

void rot_right(int arr[], int n, int d)
{
    d = d % n;

    // Original array : [1, 6, 5, 9, 8, 4]

    reverse(arr, arr + n); // [4, 8, 9, 5, 6, 1]

    reverse(arr, arr + d); // [9, 8, 4, 5, 6, 1]

    reverse(arr + d, arr + n); // [9, 8, 4, 1, 6, 5]
}

int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    int d;
    cin >> d;
    rot_right(arr, n, d);
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Question: two_sum

```
#include <bits/stdc++.h>
using namespace std;

int find_two_sum_no(int arr[], int n, int target)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = i + 1; j < n; j++)
        {
            if (arr[i] + arr[j] == target)
            {
                cout << arr[i] << " " << arr[j] << endl;
                cout << i << " " << j << endl;
            }
        }
    }
}

// better solution than brute force
string read(int n, vector<int> book, int target)
{
    // This has tc : O(N log N)
    map<int, int> mpp;
    for (int i = 0; i < n; i++)
    {
        int a = book[i];
        int more = target - a;
        if (mpp.find(more) != mpp.end())
        {
            return "YES"; // return (mpp[more], i) for index
        }
        mpp[a] = i;
    }
    return "NO";
}

// optimal approach without using map. this is just for type one(whether there are elements that sum up to target, r
string read_optimal(int n, vector<int> book, int target)
{
    // using two pointer approach
    // tc : O(N) + O(N log N)
    int left = 0, right = 0;
    sort(book.begin(), book.end());
    while (left < right)
    {
        int sum = book[left] + book[right];
        if (sum == target)
        {
            return "YES";
        }
        else if (sum < target)
            left++;
        else
            right--;
    }
    return "NO";
}

int main()
{ /*
     int n;
     cin >> n;
     int arr[n];                 // sc : O(1)
     for (int i = 0; i < n; i++) // tc : O(N)
     {
         cin >> arr[i];
     }

     int target;
     cin >> target;

     cout << find_two_sum_no(arr, n, target) << endl;
     */

    // ---------- Static input ----------
    vector<int> static_books = {3, 1, 4, 6, 5};
    int static_target = 10;
    int static_n = static_books.size();
    cout << "Static Input Result: " << read(static_n, static_books, static_target) << endl;

    // // ---------- User input ----------
    // int n, target;
    // cout << "\nEnter number of elements: ";
    // cin >> n;

    // vector<int> user_books(n);
    // cout << "Enter the book values: ";
    // for (int i = 0; i < n; i++) {
    //     cin >> user_books[i];
    // }

    // cout << "Enter the target sum: ";
    // cin >> target;

    // string result = read(n, user_books, target);
    // cout << "User Input Result: " << result << endl;

    // return 0;
}

--------------------------------------------------------------------------------

Question: union_sorted_arrays

#include <bits/stdc++.h>
using namespace std;

vector<int> find_union_sorted_arrays(int n1, int n2, vector<int> arr1, vector<int> arr2)
{
    set<int> st;

    // total t.c. for this for loop : O(N1 log n)
    for (int i = 0; i < n1; i++) // O(N1)
    {
        st.insert(arr1[i]); // log n, where n is size of set
    }

    // total t.c. for this for loop : O(N2 log n)
    for (int i = 0; i < n2; i++) // O(N2)
    {
        st.insert(arr2[i]); // log n, where n is size of set
    }

    // O(n1+n2)
    vector<int> union1;
    for (auto it : st)
    {
        union1.push_back(it);
    }
    return union1;
}

vector<int> sortedArray(vector<int> a, vector<int> b)
{
    // t.c : O(N1+N2)
    // S.C : O(N1+N2)

    int n1 = a.size();
    int n2 = b.size();
    int i = 0;
    int j = 0;
    vector<int> unionArr;
    while (i < n1 && j < n2)
    {
        if (a[i] <= b[j])
        {
            if (unionArr.size() == 0 || unionArr.back() != a[i])
            {
                unionArr.push_back(a[i]);
            }
            i++;
        }
        else
        {
            if (unionArr.size() == 0 || unionArr.back() != b[j])
            {
                unionArr.push_back(b[j]);
            }
            j++;
        }
    }

    while (j < n2)
    {
        if (unionArr.size() == 0 || unionArr.back() != b[j])
        {
            unionArr.push_back(b[j]);
        }
        j++;
    }

    while (i < n1)
    {
        if (unionArr.size() == 0 || unionArr.back() != a[i])
        {
            unionArr.push_back(a[i]);
        }
        i++;
    }

    return unionArr;
}

int main()
{
    int n1, n2;
    cin >> n1 >> n2; // Get sizes of both arrays
    vector<int> arr1(n1), arr2(n2);

    // Read the elements of the first array
    for (int i = 0; i < n1; i++)
    {
        cin >> arr1[i];
    }

    // Read the elements of the second array
    for (int i = 0; i < n2; i++)
    {
        cin >> arr2[i];
    }

    // Get the union of both arrays
    vector<int> union_result = sortedArray(arr1, arr2);

    for (int i : union_result)
    {
        cout << i << " ";
    }

    return 0;
}
```

Question: zeros_to_end

```
#include <bits/stdc++.h>
using namespace std;

void zero_to_end(int arr[], int n)
{
    int temp[n];
    // int nonze = temp.size(); this will work for vectors not arrays
    int nonze = 0;
    for (int i = 0; i < n; i++)
    {
        if (arr[i] != 0)
        {
            temp[nonze] = arr[i];
            nonze++;
        }
    }

    for (int i = 0; i < nonze; i++)
    {
        arr[i] = temp[i];
    }

    for (int i = nonze; i < n; i++)
    {
        arr[i] = 0;
    }
}

vector<int> zero_end_vec(int n, vector<int> a)
{

    // step 1
    vector<int> temp;
    for (int i = 0; i < n; i++) // O(N)
    {
        if (a[i] != 0)
        {
            temp.push_back(a[i]);
        }
    }

    // step 2
    int nonZero = temp.size();
    for (int i = 0; i < nonZero; i++) // O(X) if X non-zero nos. are present
    {
        a[i] = temp[i];
    }

    // step 3
    for (int i = nonZero; i < n; i++) // O(N-X) // coz X non zero nos., so n-x zeros
    {
        a[i] = 0;
    }
    return a;

    // Total t.c. : O(N) + O(X) + O(N-X) = O(2N)
    // S.C. : O(N) for n sized array + O(X) for non zero numbers
    // worst case in s.c. : when all numbers in array is non zero, so total tc : O(N) + O(N) = O(2N)
}

void zero_to_end_optimal(int arr[], int n)
{
    // step 1
    int j = -1;
    for (int i = 0; i < n; i++) // O(x) for non zero nos
    {
        if (arr[i] == 0)
        {
            j = i;
            break;
        }
    }

    // if(j == -1) return arr; no non zero nos. present

    // step 2
    for (int i = j + 1; i < n; i++) // remaining array i.e. O(n-x)
    {
        if (arr[i] != 0)
        {
            swap(arr[i], arr[j]);
            j++;
        }
    }

    // Total tc : O(X) + O(N-X) = O(N)
    // s.c. : 0(1) just modifying the given array, nothing extra used
}

int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    zero_to_end_optimal(arr, n);
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Question: occurrence_char

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    string s;
    cin >> s;

    // pre compute
    int hash[26] = {0}; // this is when only small alphabets are included

    // when capital letters are included too, then
    // int hash[256] = {0};
    // everything that was subtracted when arr size was 26, is removed when capital letters are included
    // bcoz then size of array becomes 356 and all alphabets big and small are included with no need to calculate in

    // so if only small letters, then c - 'a' // The line below uses this approach where we subtract 'a' from a char
    // This works because 'a' has an ASCII value of 97, and subtracting it gives us 0 for 'a', 1 for 'b', ..., and 2
    // This technique maps the 26 lowercase English letters directly to the indices of the hash array.

    // so if only big letters, then c - 'A'
    // For uppercase letters, we would subtract 'A' from the character to get the index (i.e., 'A' maps to 0, 'B' to
    // However, in this case, we're only using lowercase letters, so we do not need this for the current implementat

    // else 256
    // This would apply if we had an array of size 256 to store the counts of all possible ASCII characters.

    for (int i = 0; i < s.size(); i++)
    {
        hash[s[i] - 'a']++; // Increment the count for the character in the hash array.
        // The expression `s[i] - 'a'` maps the character to the corresponding index in the `hash` array.
        // For example, for character 'a', 'a' - 'a' = 0, so hash[0] will be incremented.
        // This ensures that we store the frequency of each character in the string efficiently.

        // hash[s[i]];
    }

    int q;
    cin >> q;
    while (q--)
    {
        char c;
        cin >> c;
        cout << hash[c - 'a'] << endl;
        // cout << hash[c] << endl;
    }
    return 0;
}
```

Question: occurrence_integers

```
#include <iostream>
using namespace std;

int main()
{
    int n;
    // enter num of elements to be included in array
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        // enter value of elements of array
        cin >> arr[i];
    }

    // precompute
    int hash[13] = {0};
    for (int i = 0; i < n; i++)
    {
        hash[arr[i]] += 1;
    }

    int q;
    // enter the amount of number for which query needs to be done
    cin >> q;
    while (q--)
    {
        int number;
        // enter value of numbers for querying
        cin >> number;
        cout << hash[number] << endl;
    }
    return 0;
}

--------------------------------------------------------------------------------

Question: occurr

#include <bits/stdc++.h>
using namespace std;

int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    // pre compute
    map<int, int> mpp;
    for (int i = 0; i < n; i++)
    {
        // time complexity in map
        // functions like storing and fetching takes O(log N) where N is num of elements in map, in best, avg and wo
        // im map. any data structure can be a key, like pair<int,int> too
        mpp[arr[i]]++;

        // in unordered map, only single valued data structures can be a key
        // if unordered map is used here in place of map,
        // then t.c. : for storing and fetching -> O(1) i.e. constant time in best and average cases
        // but in worst case it takes O(N) where N is num of elements in map
        // the worst case happens due to internal collisions

        // so make use of unordered in most and mostly all cases, but if it gives time limit exceeded error, then sw
    }

    // iterate in the map
    for (auto it : mpp)
    {
        cout << it.first << "->" << it.second << endl;
    }
    int q;
    cin >> q;
    while (q--)
    {
        int number;
        cin >> number;
        cout << mpp[number] << endl;
    }
    return 0;
}
```

Question: basics

```
#include <iostream>
using namespace std;

void printFiveXName(int i, int n)
{
    // T.C. : O(n) and S.C. :  O(n)

    // i = 1;
    // cin >> n;
    if (i > n)
        return;
    cout << "name" << endl;
    printFiveXName(i + 1, n);
    return;
}

void print1toN(int i, int n)
{
    // int N;
    // cin >> N;
    if (i > n)
        return;
    cout << i << endl;
    print1toN(i + 1, n);
    return;
}

void printNto1(int n, int i)
{
    if (i > n)
        return;
    cout << n << endl;
    printNto1(n - 1, i);
    return;
}

void OnetoNusingBack(int i, int n)
{
    if (i < 1)
        return;
    OnetoNusingBack(i - 1, n);
    cout << i << endl;
}

void NtoOneusingBack(int i, int n)
{
    if (i < 1)
        return;
    NtoOneusingBack(i - 1, n);
    cout << i << endl;
}

void sumOfN(int i, int n, int &sum)
{

    if (i > n)
        return;
    sum += i;
    sumOfN(i + 1, n, sum);
}

// Parametrized way
```

```
void hisSumToN(int i, int sum)
{
    if (i < 1)
    {
        cout << sum << endl;
        return;
    }
    hisSumToN(i - 1, sum + i);
}

// functional way
int funcSum1toN(int n)
{
    if (n == 0)
    {
        return 0;
    }
    return n + funcSum1toN(n - 1);
}

int fibo(int n)
{
    if (n == 0)
    {
        return 0;
    }
    else if (n == 1)
    {
        return 1;
    }
    else
    {
        return fibo(n - 1) + fibo(n - 2);
    }
}

int factorialN(int n)
{
    if (n == 0 || n == 1)
    {
        return 1;
    }
    return n * factorialN(n - 1);
}

void print(int i, int n)
{
    // Relying on the function call stack to stop recursion
    try
    {
        print(i + 1, n);  // Go till i > n (no base case)
        cout << i << " "; // Print during backtracking
    }
    catch (...)
    {
        // Catch any exception (not mandatory here)
    }
}

int main()
{
    // sumOfN - commented part
    // int sum = 0;
    // sumOfN(1, 6, sum);
    // cout << "sum is : " << sum << endl;

    // hisSumToN(6, 0);

    cout << factorialN(5);
    return 0;
}

/* code that improved mistakes and logic where i was lacking

The issue with your code lies in how the variable `sum` is being handled within the recursive function.

### Problem Explanation:
```

1. **Local Scope of `sum`:**

   * The variable `sum` is declared inside the function `sumOfN`, which means it is a **local variable**. Each time
   * In other words, you're trying to accumulate the sum, but since the variable is local to each call, it resets ev

2. **Recursive Calls Don't Accumulate `sum`:**

   * You are not returning or passing `sum` to the next recursive call, so the value of `sum` is lost as each recurs
   * You need to accumulate the sum across different recursive calls. This requires either:

     * Passing the `sum` variable along with each recursive call.
     * Returning `sum` from each recursive call and combining it with the value from the next call.

How to Fix It:

You should modify the function to accumulate `sum` properly, either by passing it as a parameter or by using a retur

Solution 1: Pass `sum` as a parameter:

```cpp
#include <iostream>
using namespace std;

void sumOfN(int i, int n, int& sum)
{
    if (i > n)
        return;

    sum += i;  // Add i to sum

    sumOfN(i + 1, n, sum);  // Recursively call with the updated sum
}

int main()
{
    int sum = 0;
    sumOfN(1, 6, sum);  // Pass the initial sum (0) by reference
    cout << "Sum: " << sum << endl;  // Print the final sum
    return 0;
}
```

Explanation:

* `sum` is now passed by reference (`int& sum`), so when it is updated inside `sumOfN`, the changes persist across r
* The final sum is computed and printed in `main()` after the recursive calls finish.

Output:

```
Sum: 21
```

Solution 2: Return the sum (alternative approach):

Alternatively, you could return the sum from each recursive call and combine the results. Here's the code for that:

```cpp
#include <iostream>
using namespace std;

int sumOfN(int i, int n)
{
    if (i > n)
        return 0;  // Base case: return 0 when i > n

    return i + sumOfN(i + 1, n);  // Return the sum of i and the result of the next recursive call
}

int main()
{
    int sum = sumOfN(1, 6);  // Call the recursive function and store the result
    cout << "Sum: " << sum << endl;  // Print the final sum
    return 0;
}
```

Explanation:

* Instead of using a reference to `sum`, we return the sum from each recursive call. Each recursive call computes it
* When `i > n`, we return `0` as the base case, and the recursion unwinds by adding `i` to the returned values.

Output:

```
Sum: 21
```

Conclusion:

The main issue was that the `sum` variable was not accumulating across recursive calls due to its local scope. To fi

*/

Question: functional

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

void swapArr1p(int i, int a[], int n)
{
    // *i = 0;
    // if (*i >= n/2) return;
    // (i, n-i-1);

    if (i >= n / 2)
        return;
    swap(a[i], a[n - i - 1]);
    swapArr1p(i + 1, a, n);
}

bool check_pal(int i, string &s)
{
    if (i >= s.size() / 2)
        return true;
    if (s[i] != s[s.size() - i - 1])
        return false;
    return check_pal(i + 1, s);
}

int main()
{
    /*  Swap array using one variable

    int n;
    cin >> n;
    int a[n];
    for (int i = 0; i < n; i++)
        cin >> a[i];
    swapArr1p(0, a, n);
    for (int i = 0; i < n; i++)
        cout << a[i] << " ";
    return 0;

    */

    string s = "abhi";
    cout << check_pal(0, s);
    return 0;
}
```

Question: multiple_calls

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;
```

```
int f(int n)
{
    /* here for every step 2 recursion calls were made, so this is almost 2 to the power n.

    Why Time Complexity is Almost O(2n)O(2n):

    Exponential Growth of Recursive Calls:
    For each call, we make two additional recursive calls: one for f(n-1) and another for f(n-2). This creates a bin
    The number of calls grows exponentially as the input n increases. For instance, if n = 5, the number of calls in

    2. **Why Almost \( O(2^n) \) and Not Exactly \( O(2^n) \)**:
 - **Exact Exponential Growth**: If each level of recursion were to always produce two new calls, the recursion tree
 - However, because of overlapping subproblems (e.g., `f(2)` is called multiple times), the actual number of calls w

 - In fact, without memoization or dynamic programming, you’ll end up recalculating values multiple times. For examp

 so this is an example of exponential time complexity
    */

    if (n <= 1)
        return n;
    int last, slast;
    last = f(n - 1);
    slast = f(n - 2);
    return last + slast;
}

int main()
{
    cout << f(0) << " " << f(1) << " " << f(2) << " " << f(3) << " " << f(4);
}
```

Question: basic_selection_sort

```
#include <bits/stdc++.h>
using namespace std;

void selection_sort(int arr[], int n)
{
    for (int i = 0; i <= n - 2; i++)
    {
        int mini = i;
        for (int j = i; j <= n - 1; j++)
        {
            if (arr[j] < arr[mini])
            {
                mini = j;
            }
        }
        int temp = arr[mini];
        arr[mini] = arr[i];
        arr[i] = temp;
    }
}

int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    selection_sort(arr, n);
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Question: bubble_sort

```
#include <bits/stdc++.h>
using namespace std;

void bubb_sort(int arr[], int n)
{
    for (int i = n - 1; i >= 1; i--)
    {
        int didSwap = 0;
        for (int j = 0; j <= i - 1; j++) // here j is going till i-1 bcoz if we went till i instead of i-1
        // then last element would have no next element to compare itself with
        // and it would have given a runtime error bcoz we are trying to access an index which is not present
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j + 1];
                arr[j + 1] = arr[j];
                arr[j] = temp;
                didSwap = 1;
            }
        }

        // here didSwap variable is introduced to stop the loop if no swaps happened in the previous case.
        // This reduces the original time complexity from O(N^2) to O(N) which is the best t.c. for bubble sort
        cout << didSwap << endl;
        if (didSwap == 0)
        {
            break;
        }
    }
}

int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    bubb_sort(arr, n);
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Question: insertion_sort

```
#include <bits/stdc++.h>
using namespace std;

void inser_sort(int arr[], int n)
{
    // best t.c. -> O(N) already sorted array
    // avg, worst t.c. -> O(N^2)
    for (int i = 0; i < n; i++)
    {
        int swaps = 0;
        int j = i;
        while (j > 0 && arr[j - 1] > arr[j])
        {
            int temp = arr[j - 1];
            arr[j - 1] = arr[j];
            arr[j] = temp;
            j--;
            swaps++;
        }
        cout << swaps << endl;
    }
}

int main()
{
    int n;
    cin >> n;
    int arr[n];
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    inser_sort(arr, n);
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    return 0;
}

/*

Initial Array: 46 13 24 9 20 52
i = 0 (46)

    Already the first element → no swaps
    ■ swaps = 0

i = 1 (13)

    Compare with 46 → swap → [13, 46, 24, 9, 20, 52]
    ■ swaps = 1

i = 2 (24)

    Compare with 46 → swap

    Compare with 13 → no swap
    → [13, 24, 46, 9, 20, 52]
    ■ swaps = 1

i = 3 (9)

    9 < 46 → swap

    9 < 24 → swap

    9 < 13 → swap
    → [9, 13, 24, 46, 20, 52]
    ■ swaps = 3

i = 4 (20)

    20 < 46 → swap

    20 < 24 → swap

    20 > 13 → stop
    → [9, 13, 20, 24, 46, 52]
    ■ swaps = 2

i = 5 (52)

    Already in correct position
    ■ swaps = 0

Final Output:

0
1
1
3
2
0
9 13 20 24 46 52

--------------------------------------------------------------------------------

Great! Let's do both:

---

## ■ 1. **Count Comparisons (Instead of Swaps)**
```

In **insertion sort**, a **comparison** happens every time we check `arr[j - 1] > key`.

So for each element, we can count how many times it compares itself to prior elements — this gives you a better idea

---

■ 2. **Use Real Insertion-style Shifting (No Swaps)**

In actual insertion sort, instead of swapping on every comparison (which is inefficient), we:

1. Store the `key` (current element).
2. Shift all larger elements one position ahead.
3. Insert the `key` in the correct spot.

■ Modified Code (Both Improvements)

```cpp
#include <bits/stdc++.h>
using namespace std;

void insertion_sort(int arr[], int n)
{
    for (int i = 1; i < n; i++) // Start from 1, since 0th element is trivially sorted
    {
        int key = arr[i];
        int j = i - 1;
        int comparisons = 0;

        // Shift elements greater than key to the right
        while (j >= 0 && arr[j] > key)
        {
            arr[j + 1] = arr[j];
            j--;
            comparisons++;
        }

        // One final comparison that failed (optional: count it)
        if (j >= 0) comparisons++;

        arr[j + 1] = key;

        cout << "Pass " << i << ": comparisons = " << comparisons << endl;
    }
}
```

■ Input:

```plaintext
6
46 13 24 9 20 52
```

■ Output (for comparisons):

```
Pass 1: comparisons = 1
Pass 2: comparisons = 2
Pass 3: comparisons = 3
Pass 4: comparisons = 3
Pass 5: comparisons = 1
```

■ Final Sorted Array:

```plaintext
9 13 20 24 46 52
```

```
## ■ Key Takeaways:

* Comparisons better reflect algorithm's performance.
* Actual insertion sort uses **shifting**, not repeated swapping — more efficient.
* Total comparisons here: `1 + 2 + 3 + 3 + 1 = 10` — not bad for 6 elements.

---

Would you like to visualize this as a step-by-step table or compare it with selection/merge sort side-by-side?

*/

--------------------------------------------------------------------------------

Question: merge_sort

#include <bits/stdc++.h>
using namespace std;

void merge(vector<int> &arr, int low, int mid, int high)
{
    vector<int> temp; // Temporary array to store merged result
    // Space Complexity: O(n), where n = high - low + 1

    // Left subarray -> [low....mid]
    // Right subarray -> [mid+1....high]

    int left = low;
    int right = mid + 1;

    // Merge the two sorted halves
    while (left <= mid && right <= high)
    {
        // Time Complexity (in total for the entire merge step across the array): O(n)
        if (arr[left] <= arr[right])
        {
            temp.push_back(arr[left]);
            left++;
        }
        else
        {
            temp.push_back(arr[right]);
            right++;
        }
    }

    // Copy any remaining elements from the left subarray
    while (left <= mid)
    {
        temp.push_back(arr[left]);
        left++;
        // Worst case: O(n) if all elements are in left subarray
    }

    // Copy any remaining elements from the right subarray
    while (right <= high)
    {
        temp.push_back(arr[right]);
        right++;
        // Worst case: O(n) if all elements are in right subarray
    }

    // Copy merged elements back into original array
    for (int i = low; i <= high; i++)
    {
        arr[i] = temp[i - low];
        // Time Complexity: O(n)
    }

    // Total Time Complexity of merge() = O(n), where n = high - low + 1
    // Total Space Complexity of merge() = O(n)
}

void mS(vector<int> &arr, int low, int high)
{
    // Base case: when only one element is left
    if (low == high)
```

```
        return;

    // Calculate the middle point
    int mid = (low + high) / 2;

    // Recursively sort the left half
    mS(arr, low, mid);

    // Recursively sort the right half
    mS(arr, mid + 1, high);

    // Merge the sorted halves
    merge(arr, low, mid, high);

    // Time Complexity per level: O(n) (from merge)
    // Number of levels (recursive depth): O(log n)
    // So, total Time Complexity: O(n log n)

    // Space Complexity: O(log n) due to recursive call stack (implicit)
    // Plus O(n) for temp arrays in each merge
    // So, overall space:
    // - Auxiliary/Recursive stack: O(log n)
    // - Temporary arrays: O(n) (not per call, total across all levels)
}

void mergeSort(vector<int> &arr, int n)
{
    mS(arr, 0, n - 1);
    // This function is just a wrapper, no additional complexity added
}

int main()
{
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;

    vector<int> arr(n);

    cout << "Enter " << n << " elements:\n";
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i]; // Read each element from user input
    }

    // Call mergeSort to sort the array
    mergeSort(arr, n);

    cout << "Sorted array: ";
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " "; // Output each element followed by a space
    }
    cout << endl;

    return 0;
}
```

Question: quick_sort

```
#include <bits/stdc++.h>
using namespace std;

int mypartition(vector<int> &arr, int low, int high)
{
    int pivot = arr[low];
    int i = low;
    int j = high;

    // Explanation of partition:
    // This function rearranges elements such that:
    // - All elements <= pivot are on the left side
    // - All elements > pivot are on the right side
    // Finally, pivot is placed at its correct sorted position,
    // and the index of the pivot is returned.

    while (i < j)
    {
        // Move i forward while arr[i] is less than or equal to pivot
        // and i doesn't go out of bounds.
        while (arr[i] <= pivot && i <= high - 1)
        {
            i++;
        }
        // Move j backward while arr[j] is greater than pivot
        // and j doesn't go out of bounds.
        while (arr[j] > pivot && j >= low + 1)
        {
            j--;
        }
        // Swap elements at i and j if i < j to move misplaced elements.
        if (i < j)
        {
            swap(arr[i], arr[j]);
        }
    }

    // Place pivot at its correct position.
    swap(arr[low], arr[j]);

    // Return the pivot index for further recursive calls.
    return j;
}

void quickSort(vector<int> &arr, int low, int high)
{
    // QuickSort recursively sorts subarrays divided by the pivot position.

    if (low < high)
    {
        int loc = mypartition(arr, low, high);
        quickSort(arr, low, loc - 1);
        quickSort(arr, loc + 1, high);
    }
}

int main()
{
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;

    vector<int> arr(n);
    cout << "Enter " << n << " elements:\n";
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    quickSort(arr, 0, n - 1);

    cout << "Sorted array: ";
    for (int num : arr)
        cout << num << " ";
    cout << endl;

    return 0;
}

/*
---------------------------------------------
Time Complexity Analysis of QuickSort Algorithm
---------------------------------------------

1. Partition Function Complexity:
   - The partition function traverses the subarray between indices 'low' and 'high' at most once.
   - The pointers 'i' and 'j' move towards each other and each element is examined at most once during partitioning.
   - Therefore, the time complexity of partitioning a subarray of size 'm' is O(m).

2. QuickSort Recursive Complexity:
   - QuickSort recursively calls itself on two subarrays divided by the pivot position returned from partition.
   - In the best and average case, the pivot splits the array into two roughly equal halves.

   Best/Average Case:
   - At each level of recursion, the array is split approximately in half.
   - Number of levels of recursion = O(log n).
   - At each level, the partition function runs over all elements currently in the subarrays.
   - Summing the work at each level gives: O(n) + O(n) + ... (log n times) = O(n log n).

   Worst Case:
   - Worst case occurs when the pivot is always the smallest or largest element (e.g., when the input is already sor
   - The partition divides the array into one subarray of size 1 and another of size n-1.
   - Number of recursion levels becomes O(n).
   - At each level, partition takes O(n), O(n-1), O(n-2), ..., total O(n^2).
   - Hence, worst-case time complexity is O(n^2).

Summary:
- Best/Average case: O(n log n)
- Worst case: O(n^2)

---------------------------------------------
Space Complexity Analysis
---------------------------------------------

1. Space Complexity for Partition Function:
   - The partition function uses constant extra space (only a few integer variables).
   - So, space complexity of partition function itself is O(1).

2. Space Complexity for QuickSort:
   - QuickSort is an in-place sorting algorithm, meaning it sorts the array without allocating extra arrays.
   - However, recursive calls consume stack space.
   - The depth of recursion determines the space complexity.

   Best/Average Case:
   - When partition divides the array evenly, the depth of recursion is O(log n).
   - Therefore, space complexity due to recursion stack is O(log n).

   Worst Case:
   - When partition is unbalanced (like always picking smallest/largest element as pivot), recursion depth becomes O
   - Thus, worst-case space complexity is O(n) due to the stack.

Summary:
- Best/Average case space complexity: O(log n) (due to recursion)
- Worst case space complexity: O(n) (due to recursion stack)

---------------------------------------------
Note:
- This implementation always picks the first element as pivot.
- Pivot selection strategy affects the average and worst-case time complexity.
- Randomized pivot or median-of-three pivot selection often improves average case performance and avoids worst case.

*/


--------------------------------------------------------------------------------
```